Eitan Frachtenberg
Uwe Schwiegelshohn (Eds.)

Job Scheduling Strategies for Parallel Processing

LNCS 6253

15th International Workshop, JSSPP 2010
Atlanta, GA, USA, April 2010
Revised Selected Papers

Springer

Lecture Notes in Computer Science 6253

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Eitan Frachtenberg Uwe Schwiegelshohn (Eds.)

Job Scheduling Strategies for Parallel Processing

15th International Workshop, JSSPP 2010
Atlanta, GA, USA, April 23, 2010
Revised Selected Papers

Springer

Volume Editors

Eitan Frachtenberg
Facebook, 475 Brannan St.
San Francisco, CA, 94107, USA
E-mail: etc_26@yahoo.com

Uwe Schwiegelshohn
Robotics Research Institute
Section Information Technology
TU Dortmund University
Otto-Hahn-Str. 8
44227 Dortmund, Germany
E-mail: uwe.schwiegelshohn@udo.edu

Library of Congress Control Number: 2010937281

CR Subject Classification (1998): C.2.4, D.4, C.2, D.2, H.3, I.6

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-642-16504-4 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-16504-7 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

This volume contains the papers presented at the 15^{th} workshop on Job Scheduling Strategies for Parallel Processing that was held in Atlanta (GA), USA, on April 23, 2010 in conjunction with the IEEE International Parallel Processing Symposium 2010.

This year 18 papers were submitted to the workshop. All submitted papers went through a complete review process, with the full version being read and evaluated by an average of four reviewers. We would like to especially thank the program committee members and additional referees for their willingness to participate in this effort and their excellent, detailed reviews:

Henri Casanova, Peter A. Chronz, Walfredo Cirne, Julita Corbalan, Arash Deshmeh, Dick Epema, Dror G. Feitelson, Allan Gottlieb, Rajkumar Kettimuthu, Virginia Lo, Kuan Lu, Vicent Matossian, Jose E. Moreira, Bill Nitzberg, Elizeu Santos-Neto, Angela C. Sodan, Mark S. Squillante, Dan Tsafrir, Philipp Wieder, and Ramin Yahyapour.

The papers in this volume show a prolific growth in the areas of applicability for parallel scheduling. Together with the more common scheduling aspects (such as cluster and Grid scheduling, workload analysis, metrics, quality of service, and task scheduling), these papers increasingly discuss more recent problems and applications, such as virtualized environments, many-core processors, DNA sequencing, and Hadoop. This volume also includes a paper that summarizes Dan Tsafrir's work on understanding the role of user estimates in job scheduling evaluations. His insights, which were presented in this workshop's keynote, are quite instructive and lead to the conclusion that accurate user estimates are indeed better for efficient scheduling. Although this conclusion may sound intuitive, it is actually contradictory to previous studies that found inaccurate estimates to improve scheduler performance. Following his analysis, Dan also suggests practical ways to deal with estimate inaccuracy for realistic job scheduler evaluations.

One of the stated goals of the JSSPP workshop is to explore the application of traditional scheduling topics to novel scenarios. A good example of such topics is task and graph scheduling, which until last year was largely outside the scope of JSSPP. Recent technologies are reviving interest in this topic, as was discussed last year in the context of workflow jobs in a Grid environment (in a paper by Gong et al.), and this year in the context of the emerging multi-core/multi-threaded processors. Xia, Prasanna, and Li apply ideas from dynamic load balancing and hierarchical thread grouping to the contemporary architecture of the Sun Niagara processor with its 64 hardware threads. A different thread-level scheduling aspect for many-core architectures is introduced in the paper by Yamada and Kusakabe. Here, a modification to the operating system is suggested to permit multi-threaded applications to aggregate time and space resources for improved cache efficiency. In another example of applying

traditional scheduling to newer problems, the paper by Saule, Bozdag, and Catalyurek shows how a contemporary DNA sequencing workload can benefit – in terms of reduced slowdown – from the application of earliest-deadline-first to moldable-job scheduling.

Two other papers combine traditional scheduling techniques with modern technological challenges and capabilities. For the former, the paper by Klusáček and Rudová? addresses a deficiency in many of the workload traces used for job scheduling simulations: the lack of complete system and workload information, including machine characteristics and failures. Adding these data to a trace (synthetically or with actual collected data, in the case of MetaCentrum's trace) can significantly alter the results of a scheduler evaluation. For the latter, the paper by Verboven, Vanmechelen, and Broeckhove introduces a scheduling scheme for virtual machines, where the workload has mixed high and low quality-of-service (QoS) requirements. The scheduler takes advantage of the relative ease that virtualization offers for job preemption to let overbooked low-priority jobs fill in for underutilized resources, without interrupting high-priority jobs. Overbooking is also explored in the paper by Birkenheuer, Brinkmann, and Karl, extending their work from the previous workshop. Here, the authors integrate a statistical risk assessment module to their Grid/Cloud scheduler that uses automated run-time predictions to overbook backfilled "gaps", while still minimizing the economic penalties to the system owner from missed deadlines.

The economics of Grid scheduling and QoS were in fact a major theme in this year's papers, and played a minor part in several other papers. The paper by Sandholm and Lai explores a dynamic resource allocation scheduler for the popular Hadoop environment, in which users receive a share of computational resources that is proportional to their bid. On a related vein, in Ding's contribution, a greedy double-auction mechanism to dynamically price resources is simulated, backed by a theoretical analysis. Xiong's paper on the other hand assumes static pricing per QoS level, and proposes a resource allocation approach to minimize the total cost to the application, again offering a theoretical treatment of the optimization problem. Similarly, in the paper by Takefusa, Nakada, Kudoh, and Tanaka, a linear-programming method is used to co-allocate processing and networking resources to Grid applications, based on an actual system's problem statement. And last but not least, in the metascheduler proposed by Fölling, Grimme, Lepping, and Papaspyrou, different sites can "lease;; underutilized resources to heavily loaded sites that request them.

A few years ago, we described the transition of the classic job scheduling paradigm for parallel processing based on the proliferation of new technologies like many-core architectures and Grid or Cloud computing. Today, we can observe that these technological advances produce significant changes in the usage patterns. Due to many-core architectures, the exploitation of parallelism is not restricted any more to a few high performance applications. Similar to the situation during the last decade of the previous century when every application exploited superscalar processors, applications are now expected to make efficient use of multiple cores. This of course results in new challenges for job scheduling.

Furthermore, Grid and Cloud computing provides affordable processing power to many users, leading to new applications if there are efficient resource management systems. These resource management systems will consist of multiple layers related to users, resource providers, and domain managers. However, the allocation of tasks to these layers is still under investigation. Therefore, we strongly believe that research in the field of this workshop will remain interesting and challenging in the years to come.

The proceedings of previous workshops are available from Springer as LNCS volumes 949, 1162, 1291, 1459, 1659, 1911, 2221, 2537, 2862, 3277, 3834, 4376, 4942, and 5798. Since 1995 these volumes have also been available online.

June 2010

Eitan Frachtenberg
Uwe Schwiegelshohn

Organization

Workshop Organizers

Eitan Frachtenberg	Facebook
Uwe Schwiegelshohn	TU Dortmund University

Program Committee

Henri Casanova	University of Hawaii at Manoa
Walfredo Cirne	Google
Julita Corbalan	Technical University of Catalunya
Dick Epema	Delft University of Technology
Dror Feitelson	The Hebrew University
Allan Gottlieb	New York University
Rajkumar Kettimuthu	Argonne National Lab
Virginia Lo	University of Oregon
Jose Moreira	IBM Thomas J. Watson Research Center
Bill Nitzberg	Altair Engineering, Inc.
Angela Sodan	University of Windsor
Mark Squillante	IBM Thomas J. Watson Research Center
Dan Tsafrir	Technion
Ramin Yahyapour	TU Dortmund University

Table of Contents

Resource Provisioning in SLA-Based Cluster Computing

Kaiqi Xiong and Sang Suh

Department of Computer Science, Texas A&M University,
Commerce, TX 75429, USA
kaiqi_xiong@tamu-commerce.edu

Abstract. Cluster computing is excellent for parallel computation. It has become increasingly popular. In cluster computing, a service level agreement (SLA) is a set of quality of services (QoS) and a fee agreed between a customer and an application service provider. It plays an important role in an e-business application. An application service provider uses a set of cluster computing resources to support e-business applications subject to an SLA. In this paper, the QoS includes percentile response time and cluster utilization. We present an approach for resource provisioning in such an environment that minimizes the total cost of cluster computing resources used by an application service provider for an e-business application that often requires parallel computation for high service performance, availability, and reliability while satisfying a QoS and a fee negotiated between a customer and the application service provider. Simulation experiments demonstrate the applicability of the approach.

Keywords: Cluster computing, scheduling theory, resource provisioning, service level agreement, and percentile response time.

1 Introduction

In computer science, scheduling theory is concerned with the optimal allocation of scarce resources such as servers, processors and network links to computer service activities over time, with the objective of optimizing one or several computer performance measures (e.g., see Levner [13]). Cluster computing is excellent for parallel computation. It has become increasingly popular. The management of computing service resources is fundamental to cluster computing. The increasing pervasiveness of network connectivity and the proliferation of on demand e-business applications and services in public domains, corporate networks, as well as home environments give rise to the need for the design of appropriate service management solutions in cluster computing. Accurately predicting e-business application and scientific computation performance based on service statistics and a customer's perceived quality allows an application service provider (simply called a service provider) not only to assure quality of services but also to avoid over provisioning to meet a service level agreement (SLA).

E. Frachtenberg and U. Schwiegelshohn (Eds.): JSSPP 2010, LNCS 6253, pp. 1–15, 2010.

Job scheduling has been a fruitful area of research for many decades. It involves answers to the following questions:

1. How are jobs assigned to computing resources, such as processors and machines?
2. What orders should we use to process jobs in a single computing resource?
3. How to allocate sufficient computing resources to match the requirements of submitted jobs in terms of ensuring QoS guarantees?

The above three job scheduling questions are usually called the parallel job scheduling problem, the job sequencing problem, and the resource provisioning problem (also called the resource matching problem), respectively. While the job sequencing problem is relatively simple, the parallel job scheduling problem and the resource allocation problem are difficult to solve. Generally speaking, both are NP-hard (see Du and Leung [7]). In this paper, we focus on the resource provisioning problem that has been extensively researched over the years (see Feitelson et al. [8] and Yom-Tov and Aridor [19]). In particular, we consider the problem for avoiding the over-provisioning of computing resources. With over-provisioning, computing resources are allocated more than service request jobs actually need due to the over-determined requirements of service request jobs, which should not occur as desired by a service provider for high profits.

Yom-Tov and Aridor [19] gave an example of two machines to explain how badly over-provisioning affects machine utilization. However, if allocated computing resources fall below a certain level or are insufficient, service request jobs cannot complete to meet customer service requirements. Hence, the resource provisioning problem plays a key role in job scheduling. It is an extremely important but very challenging problem as shown in Liu et al. [12], Naik et al [16], and Yom-Tov and Aridor [19].

In this paper, we consider a resource provisioning problem in SLA-based cluster computing where a service provider processes e-business application request jobs for business customers subject to an SLA. Such request jobs often require parallel computation for high service performance, availability, and reliability. As shown in Figure 1, a customer represents a business that generates a stream of service request jobs at a specified rate to be processed by a service provider's resources according to QoS requirements and for a given fee. A service request job is transmitted to a service provider in a cluster computing system consisting of a group of cluster nodes that are linked together to support parallel computation (e.g., see Aron et al. [1], Heath et al. [10] as well as Xiao and Ni [35]). Upon the completion of a service request job at the service provider, the final result is sent back to the customer. The service provider's cluster nodes have or are a set of computing resources so that they are capable to collaborate each other in parallel for processing the service request job. Such computing resources in each cluster node may include processors and cluster servers/machines as discussed in Shin et al. [17] as well as Xiao and Ni [35]. For presentation purpose, we explicitly think the computing resource of each node as cluster servers. (Note that in this paper "computing resource" or "server" are used alternatively.)

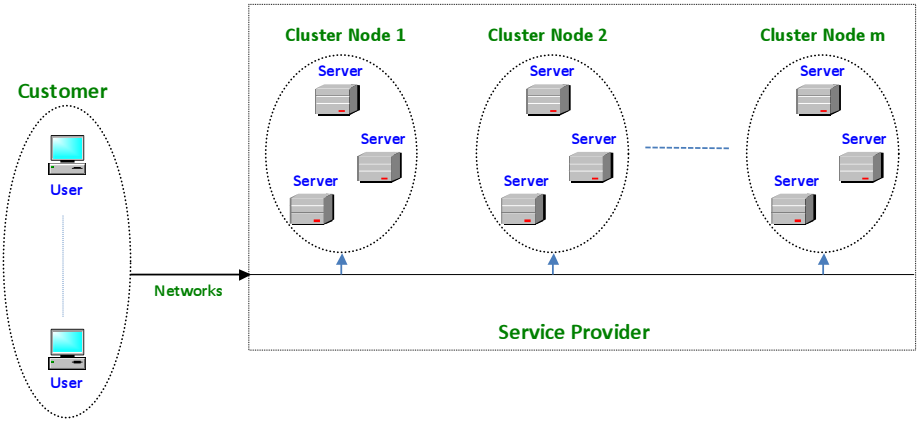


Fig. 1. Customer Service Request Jobs in SLA-based Cluster Computing

The resource provisioning problem is to minimize the overall cost of the service provider's computing resources of each node allocated to the business customer in terms of the number of servers at each cluster node while satisfying an SLA agreement. The SLA is a contract negotiated and agreed between the customer and the application provider. It defines the quality of service (QoS) and a fee. In this paper, the QoS metrics include:

1. *Percentile response time*-γ% ($0 \leq \gamma \leq 100$) of the time the response time, i.e., the time to execute a service request job, is less than a pre-defined value;
2. *Cluster utilization*-It is the percentage of the time that the cluster node is utilized.

Both of them are often called SLA performance metrics in the literature. These QoS requirements are typical metrics included in an SLA (e.g., see INTERNAP [25] as well as Martin and Nilsson [28]). As an end user of e-business applications, a customer is in general concerned about response time rather than throughput (for example, in an online business, an buyer often concerns about how soon his/her order will be processed and completed). Hence, we do not include throughput as a metric in this study. Security, reliability and survivability may be included in an SLA as well as described in Jacob [26]. We will discuss them in another paper.

In this paper, we present the resource provisioning problem by minimizing the total cost of each cluster node's computer resources required to ensure a given percentile of the response time and cluster utilization. We formulate the provisioning problem as a constrained optimization problem. By modeling a typical customer service scenario as a queueing network, we first propose an approach to computing the percentile response time of a service request job. We note that the proposed approach can be also applied to a queueing network whose cluster nodes are arbitrarily linked as long as the link can be quantified. Then,

we present an approach to solving the constrained optimization problem by calculating the computing resource of each cluster node required in each service provider's node. To the best of our knowledge, our study is the first attempt to analytically solve the resource provisioning problem under the consideration of *percentile response time and cluster utilization* for cluster computing by using a queueing network method.

The rest of the paper is organized as follows. Related work is presented in Section 2. In Section 3 we formulate the resource provisioning problem with the SLA performance metrics. In Section 4 we model the service request jobs processed in SLA-based cluster computing as a queueing network and give an approximation approach to computing the percentile response time of a customer service request job in the queueing network. We further propose an approach for solving the provisioning problem. Numerical experiments are given in Section 5. We conclude our results in Section 6.

2 Related Work

The job scheduling questions presented in Section 1 have been extensively studied. They play an important role in not only parallel computation but also other areas. Many real-world problems can be modeled as scheduling problems. For example, the relationship between jobs and computing resources is similar to the one between the following pairs: students and teachers, patients and doctors, as well as ships and docks. Only a few scheduling problems have been shown to be tractable, that is, they are solvable in polynomial time. For the remaining ones, the only way to secure optimal solutions is usually by enumerative methods, requiring exponential time (e.g., see Cook [6], Garey and Johnson [9], and Papadimitriou [15]).

Resource management including resource monitoring as well as resource matching and/or resource provisioning has been researched over many years. Feitelson et al. [8] and Yom-Tov and Aridor [19] have studied resource provisioning for job scheduling in heterogeneous server clusters. Ngubiri and Vlient [14] discussed a processor provisioning problem in multi-cluster systems. Bucur [3], Bucer and Epema [4], and Jones [11] have considered the problem of resource provisioning for Distributed ASCI Supercomputer (DAS). Bucur and Epema [4] proposed and analyzed resource provisioning approaches in different scenarios. Jones [11] focused on scheduling techniques and how they are affected by network characteristics like latencies.

In the paper, we consider a job as a stream of customer service requests in cluster computing. The problem of multiple heterogeneous resources allocated to a single job has been discussed in Liu et al. [12]. It is an one-to-many matching problem under the constraints of application specific global aggregations, for example, total memory sizes and processor capacities.

As we know, in the above literature the authors only considered the average metric value of a job stream as a performance metric. This is because an average metric value is relatively easy to calculate. However, customer is more inclined

to request a statistical bound on its response time than an average response time. Thus, we use the percentile response time as our performance metric in the paper.

Resource provisioning with the constraints of a variety of QoS metrics such as response time, cluster utilization, or packet loss rate for other computing infrastructures such as a network system have been extensively studied in the literature as well. Bouillet, et al. [20] considered a routing and resource management problem subject to the requirements of aggregate bandwidth from ingress to egress nodes. In [21], Chassot et al. dealt with a communication architecture with guaranteed end-to-end QoS in an IPv6 environment. The end-to-end QoS includes an end-to-end delay (i.e., response time). Chassot et al. only discussed and measured the maximal, minimal and average values of response time. Cao and Zegura [22] considered the bandwidth allocation scheme for an available bit rate service. In Liao and Campbell [27], a mechanism was developed with the capability of delivering capacity provisioning in an efficient manner providing quantitative delay-bounds with differentiated loss across per-aggregate service classes.

3 The Resource Provisioning Problem

In this section, we study the customer service request jobs depicted in Figure 1 where a service request job is transmitted to m cluster nodes within a service provider. For presentation purposes, we assume that each cluster node has only one type of cluster server associated with cost c_j. If they have multiple types of servers, we can decompose each cluster node into several individual sub-nodes so that each one only contains one type of servers with the same cost.

Let N_j be the number of servers at node j ($j = 1, 2, \cdots, m$). Thus, the resource provisioning problem is to minimize the overall cost of the computing resources required while satisfying SLA requirements in cluster computing. That is, the resource provisioning problem is quantified by solving for d_j in the following provisioning problem:

$$I = \min_{d_1,\cdots,d_m} (d_1\, c_1 + \cdots + d_m\, c_m) \tag{1}$$

subject to SLA constraints, where d_j represents the number of servers required in cluster node j and hence its value is 1, 2, $\cdots$, or N_j, each server associated with cost c_j. Performance and a service fee are the two most important components for a variety of SLAs in high performance computing such as cluster and grid computing to support parallel computing for business applications. In this paper, the SLA constraints include the aforementioned percentile response time and cluster utilization as well as a service fee.

As discussed in Section 1, we consider cluster utilization and the percentile of response time as the SLA performance metrics. The cluster utilization is the percentage of the time that the cluster node is utilized. It will be discussed in detail in Section 4.1. The cluster utilization within a service provider is not

observed by a customer (see Martin and Nilsson [28]). Instead, response time can be directly measured by a customer. It directly reflects service performance as stated in Martin and Nilsson [28], Paxson [30] and Padhye et al. [31].

As described earlier, in the literature, typically the average response time (or an average execution time) is used (e.g., see Martin and Nilsson [28] as well as Menasce and E. Casalicchio [29]). The average response time is heavily influenced by "outliers," which occur in almost all measurements. Therefore, although the average response time is relatively easy to calculate, it may not address the concerns of a customer. Typically, a customer is more inclined to request a statistical bound on its response time than an average response time. For instance, a customer can request that 95% of the time its response time should be less than a desired value. Hence, in this paper we are concerned with the statistical bound on the response time.

The response time is the time it takes for a service request job to be executed on the service provider's cluster nodes and then sent its completed job back to the customer. Let T be a random variable representing the response time, and let $f_T(t)$ and $F_T(t)$ be its probability and cumulative distributions pdf and CDF, respectively. Also, let T^D be the desired target response time that a customer requests and agrees with its service provider based on a fee paid by the customer. The statistical bound on the response time can be expressed by

$$F_T(t)|_{t=T^D} = \int_0^{T^D} f_T(t)\,dt \geq \gamma\% \quad (0 \leq \gamma \leq 100) \tag{2}$$

which is called *percentile response time*. This means that $\gamma\%$ of the time a service request job will be executed in less than T^D.

As an example let us consider an $M/M/1$ queue with arrival rate λ and service rate μ. The service discipline is FIFO. The steady-state probability of the system is $p_0 = 1 - \rho$, and $p_k = (1-\rho)\rho^k$, $k > 0$, where $\rho = \frac{\lambda}{\mu}$ (see Perros [32]). The response time T is exponentially distributed with the parameter $\mu(1-\rho)$, i.e., its probability distribution is given by

$$f_T(t) = \mu(1-\rho)e^{-\mu(1-\rho)t}$$

Using the definition given in (2), we have that

$$F_T(t)|_{t=T^D} = 1 - e^{-\mu(1-\rho)T^D} \geq \gamma\% \tag{3}$$

For example, to ensure that in a 95% (=$\gamma\%$) of time, customer service request jobs can be executed in T^D. It follows from (3) that

$$e^{-\mu(1-\rho)T^D} \leq 5\%$$

which is equivalent to

$$\mu \geq \lambda + \frac{\ln 20}{T^D}$$

Furthermore, the resource provisioning problem can be formulated as the following integer optimization problem.

The Resource Provisioning Problem in SLA-based Cluster Computing:

Find integers d_j $(0 \leq d_j \leq N_j;\ j = 1, 2, \cdots, m)$ in the m-dimensional provisioning problem (11) under the constraints of $I \leq C^D$, the percentile response time as expressed by (2), and the cluster utilization satisfying $\rho_j \leq \zeta_j\%$, and $\rho^{overall} \leq \zeta\%$ respectively, where C^D is a fee negotiated and agreed upon between a customer and the service provider, ρ_j is the average cluster utilization of node j, and $\rho^{overall}$ is the average cluster utilization of all the cluster nodes within the service provider. Parameters ζ_j and ζ are pre-defined values in the SLA $(j = 1, 2, \cdots, m)$.

4 The Solution of the Resource Provisioning Problem

In this section, we study a queueing network model that depicts the path that service request jobs have to follow through the cluster nodes' resources owned by the service provider described in Figure 1. The queueing model is shown in Figure 2. We refer to the queueing model as a service request job model since it depicts the computing resources used to provide computing services to respond a customer's service job requests.

The service request job model consists of a single infinite server, and m service provider's stations (or simply called nodes. In the rest of this paper, without any confusion station and node are alternatively used) numbered sequentially from 1 to m as shown in Figure 2. After a customer exits from the single infinite server, it will continue to be served at all m nodes. Upon completion of its service at the m-th node, a customer may exit the queueing network with probability α, or may return to the beginning the queueing network with probability $1 - \alpha$, which characterizes the retransmission of a service request job within the service provider, shown in Figure 2.

As seen in Figure 1, each cluster node consists of multiple servers that are linked together to support for parallel computations. The servers of each cluster node are commonly, but not always, connected to each other through fast local area networks. Cluster nodes are usually deployed to improve performance and/or availability over that of a single computer, while typically being much

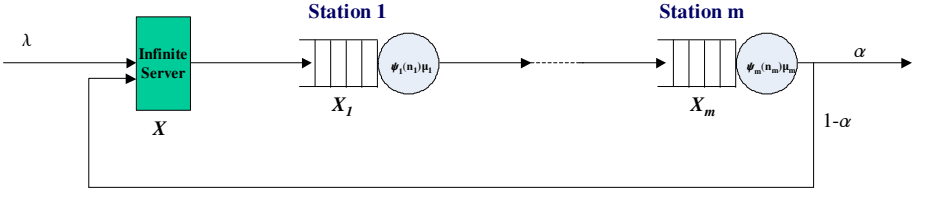


Fig. 2. A Service Request Job Model

more cost-effective than single computers of comparable speed or availability (see Luke [2]). Each cluster node has a group of linked servers to work together closely so that it is treated as a single computer in many respects. Thus, in the following discussion each service provider's cluster node is modeled as a single $G/G/1$ queue with arrival rate λ_j and service rate $\psi(d_j)\mu_j$, where $\psi(d_j)$ is a known function of d_j and depends on the configuration of servers at each node or station. It is non-decreasing and can be inverted, i.e., ψ^{-1} exists. For instance, suppose that a station represents a group of CPUs. Then, $\psi(n)$ can be seen as a CPU scaling factor for the number of CPUs from 1 to n. According to Chang [5], $\psi(n) = \xi^{\log_2 n}$, where ξ is a basic scaling factor from 1 CPU to 2. So, $\psi^{-1}(n) = \xi^{-\log_2 n}$.

Let Λ be the arrival rate generated by a customer as well as λ and λ_j be the effective arrival rates to the infinite server, respectively. The infinite server represents the total propagation delay from the first cluster node through the m-th cluster node. The first station in Figure 2 models the architecture and elements (i.e., servers) of the first cluster node in Figure 1. The j-th station in Figure 2) $(j = 2, 3, \cdots, m)$ models the architecture and elements of the j-th cluster node in Figure 1.

We have the traffic equations: $\lambda = \Lambda + (1 - \alpha)\lambda_m$ and $\lambda_j = \lambda$ that implies $\lambda_j = \lambda = \frac{\Lambda}{\alpha}$, and the utilization of each station is $\rho_j = \frac{\lambda_j}{\psi(d_j)\mu_j} = \frac{\Lambda}{\alpha\mu_j\psi(d_j)}$ $(j = 1, 2, \cdots, m)$. Note that the infinity server has the same effective arrival rate as node j. Thus, let $p(t)$ and $p_j(t, \psi(d_j)\mu_j)$ be the pdfs of response time at the infinity server and node j (these pdfs can be at least determined by a curve fitting of measurement data as discussed in Zandt [36]), and $L_X(s)$ and $L_{X_j}(s, \psi(d_j)\mu_j)$ its corresponding Laplace transform at the infinite server and node j respectively, where X is the service time at the infinite server, and X_j is the time elapsed from the moment a service request job arriving at node j to the moment it departs from the node.

4.1 An Algorithm for the Resource Provisioning Problem

In order to present our approach for solving the resource provisioning problem, we need to derive the Laplace-Stieltjes transforms (LST) of the probability distribution of the response time.

Let $T(k)$ be the response time of k-th visit at the infinite server, the first node, the second node, ..., and m-th node. Then, $T(k)$ is considered as the sum of the response time of the k-th pass at the infinite server plus the response time of the k-th pass at all the m stations:

$$T(k) = X + X_1 + X_2 + \cdots + X_m$$

where we assume that each router is independent of each other. That is, we assume that the waiting time of a service request job at a station or a node is independent of its waiting times at other stations or nodes. Then, the total response time of a service request is

$$T = \sum_{k=1}^{\infty} p(k)T(k)$$

where $p(k)$ is the steady state probability that a request will circulate k times at the infinite server and the j-th station through the computing system. $p(k)$ is determined by

$$p(k) = \alpha(1-\alpha)^{k-1}$$

Thus, the LST of the response time T is

$$L_T(s) = \sum_{k=1}^{\infty} p(k) L_X^k(s) L_{X_1}^k(s, \psi(d_1)\mu_1) \cdots L_{X_m}^k(s, \psi(d_m)\mu_m)$$

which can be re-written as follows:

$$L_T(s) = \frac{\alpha L_X(s)\, \Pi_{j=1} L_{X_j}(s, \psi(d_j)\mu_j)}{1 - (1-\alpha)\, L_X(s) \Pi_{j=1}^{m} L_{X_j}(s, \psi(d_j)\mu_j)} \tag{4}$$

where $L_X(s)$ and $L_{X_j}(s, \psi(d_j)\mu_j)$ $(j = 1, 2, \cdots, m)$ are the LST of the response time X and the response time X_j.

The probability distribution $f_T(t)$ and the cumulative distribution $F_T(t)$ of the response time T can be calculated by inverting $L_T(s)$ and $L_T(s)/s$ respectively, that is,

$$f_T(t) = L^{-1}\left(L_T(s)\right) \quad \text{and} \quad F_T(t) = L^{-1}\left(\frac{L_T(s)}{s}\right) \tag{5}$$

We observe that $f_T(t)$ and $F_T(t)$ are usually nonlinear functions of t and d_j. Hence, the resource provisioning problem is an m-dimensional linear provisioning problem subject to nonlinear constraints. In general, it is not easy to solve this problem. However, the complexity of the problem can be significantly reduced by postulating that the utilization of each node in Figure 2 should be the same for all nodes. That is, we find the optimum value of $d_1, \cdots, d_m$ such that

$$\rho_1 = \cdots = \rho_m \stackrel{def}{=} \hat{a}$$

where $\rho_j = \frac{\lambda_j}{\psi(d_j)\mu_j}$ is the average cluster utilization of the j-th node $(j = 1, 2, \cdots, m)$. This is called *the balanced condition.* (We note that in production lines, it is commonly assumed that the service stations are balanced whose further justification can be found in Xiong [18]).

We further consider the cluster utilization of the service model within the service provider's node, and derive the following result.

Proposition: The average cluster utilization of all the cluster nodes within the service provider is

$$\rho^{overall}(\hat{a}) = \frac{\hat{a}^m}{1 - (1-\alpha)\hat{a}^m} \tag{6}$$

Proof. From the structure of the queueing network, the average cluster utilization of this SLA-based cluster model within the service provider can be computed by

$$\rho^{overall}(\hat{a}) = \sum_{k=1}^{\infty} p(k) \rho_1(\hat{a}) \cdots \rho_j(\hat{a})$$

where $p(k) = \alpha(1-\alpha)^{k-1}$ and $\rho_j(\hat{a}) = \rho_j$. Due to the balanced condition, we have $\rho_j(\hat{a}) = \hat{a}$, and then easily get (6). The proof is complete.

As presented in the resource provisioning problem, the constraint of cluster utilization at each node: $\rho_j(\hat{a}_j) \leq \zeta_j\%$, and the constraint of the average cluster utilization of all the cluster nodes within the service provider: $\rho^{overall}(\hat{a}^u) \leq \zeta\%$. To ensure the cluster utilization guarantees, we require that $\hat{a}_j = \hat{a} \leq \zeta_j\%$ and $\frac{\hat{a}^m}{1-(1-\alpha)\hat{a}^m} \leq \zeta\%$. This implies that

$$\hat{a} \leq \min\left\{\zeta_1\%, \cdots, \zeta_m\%, \sqrt[m]{\frac{\zeta\%}{1+(1-\alpha)\zeta\%}}\right\} \tag{7}$$

In addition, note that $\frac{\lambda_j}{\psi(d_j)\mu_j} = \hat{a}$. Hence, $\psi(d_j) = \frac{\lambda}{\hat{a}\,\mu_j}$, i.e., $d_j = \psi^{-1}\left(\frac{\lambda}{\hat{a}\,\mu_j}\right)$ for $j = 1, 2, \cdots, m$. This implies $\sum_{j=1}^{m} c_j d_j$ reduces to a function of variable $\hat{a}$. Thus, we have the following algorithm for solving the resource provisioning algorithm.

Algorithm

a. Find $\hat{a}$ in the following minimization problem of a percentile response time and its corresponding optimum values of $d_j^{(1)}$:

$$\hat{a}^{(1)} \leftarrow \arg\min_{\hat{a}} F_T(t)|_{t=T^D}$$

subject to the constraint: $F_T(t)|_{t=T^D} \geq \gamma\%$ at $\hat{a} = \hat{a}^{(1)}$, where $F_T(t)$ is given by (5). Then, the optimum values of $d_j^{(1)}$ for the percentile response time guarantee are given by $d_j^{(1)} = \psi^{-1}\left(\frac{\lambda_j}{\hat{a}^{(1)}\mu_j}\right)$ for $j = 1, 2, \cdots, m$.

b. Calculate $\hat{a}$ given in (7) to ensure the guarantees of cluster node utilization. Their maximal values $a_j^{(2)}$ for stations 1, 2, and 3 are computed by

$$a_j^{(2)} = \frac{\lambda_j}{\mu_j}\max\left\{(\zeta_j\%)^{-1}, \sqrt[m]{\frac{1+(1-\alpha)\zeta\%}{\zeta\%}}\right\}$$

Thus, its corresponding optimum values of $d_j^{(2)}$ are equal to $d_j^{(2)} = \psi^{-1}\left(a_j^{(2)}\right)$ for $j = 1, 2, \cdots, m$.

c. Calculate the maximum values d_j^M such that $d_j^M = \max\{d_j^{(1)}, d_j^{(2)}\}$, and then choose the optimum values of d_j are equal to d_j^M $(j = 1, 2, \cdots, m)$.

d. Check if $0 \leq d_j \leq N_j$ $(j = 1, 2, \cdots, m)$ and $I \leq C^D$ are satisfied. If yes, the obtained d_j is the optimum number of servers required at each cluster node. That is, the service provider should allocate at least d_j servers at each cluster node to ensure the SLA guarantee. Otherwise, the resource provisioning problem subject to the SLA cannot be solved. In this case, the service provider will inform the customer "We need to re-negotiate the SLA," or both.

Note that if we cannot get a solution for the resource provisioning problem using the above algorithm, then the service provider cannot execute service request jobs in the SLA-based cluster computing due to at least one of the following reasons: (i) the service provider has insufficient computing resources (i.e., μ_j, N_j, or both are too small), (ii) a pre-specific fee is too low (i.e., $I > C^D$), or (iii) at least one cluster node is over-utilized. Using these information, we may detect and debug a service provider's capacity problem, that is, the SLA needs to be re-negotiated.

In this algorithm, the run-time for Steps b, c and d have the same run-time $O(m)$. The efficiency of this algorithm is determined by the run-time for inverting the LST of the response time in Step a, which can be efficiently done as well (see Graf [24]). Let T_1 be the run-time for the inversion of the LST and T_2 be the time to find $\hat{a}^{(1)}$ except the time to invert the LST of the response time. (This is an one-dimensional minimization problem. So, generally speaking, T_2 is relatively smaller than T_1.) Thus, the total run-time for the Algorithm is $O(T_1 + T_2 + m)$.

As we see, the total run-time for the Algorithm is mainly determined by $O(T_1)$, which depends on the number of function evaluations required for each value of t that is varied in each numerical approximation method for the inversion of a Laplace transform. In our numerical experiments, it usually took a couple of minutes to complete the evaluation.

Remarks: In the above algorithm, if we require that each node has the same pre-defined ζ_j, then the constraints of $\rho_j(\hat{a}_j) \leq \zeta_j\%$ $(j = 1, 2, \cdots, m)$ reduce to the only one constraint: $\rho_1(\hat{a}_1) \leq \zeta_1\%$, due to the above proposition.

5 Numerical Experiments

In this section we demonstrate how to apply our algorithm to solve the resource provisioning problem subject to an SLA.

Clearly, our proposed method heavily depends on the computation of the inverse Laplace transform of $L_T(s)$. Many studies have been done in the past a few decades as described in Graf [24]. Since the numerical computation of an inverse Laplace transform is an ill-posed problem, no single method works for any inverse Laplace transform problem (see Graf [24]). This is because in this case there is a singular point that significantly affects the numerical computation of an inverse Laplace transform. Thus, we employed several different numerical methods for inverting a given $L_T(s)$. If two or more methods can reach about the same results, then we are confident that the derived numerical inverse Laplace transform is correct. These numerical methods include the inversion methods using Laguerre functions and Fourier functions in Graf [24], Gaussian quadrature formulas in Piessens [33], and the method by Gaver [23] and Stehfest [34]. The Laguerre method in Graf [24] and the Gaver-Stehfest method in Gaver [23] and Stehfest [34] compute more rapidly but are slightly less accurate compared to the Gaussian quadrature formulas in Piessens [33].

We consider the service request job model shown in Figure 2. For presentation purpose, we only consider a three-station model, i.e., $m = 3$. The values of

Table 1. The Values of c_1, c_2, c_3, C^D, N_1, N_2, N_3, T^D, γ, α, ζ_1, ζ_2, ζ_3, and ζ

c_1	c_2	c_3	c^D	N_1	N_2	N_3	T^D	γ	α	ζ_1	ζ_2	ζ_3	ζ
8	8	3	800	50	80	100	0.08	98	0.8	0.78	0.9	0.92	0.58

parameters c_j, C^D, N_j, $T^D = 0.08$, γ, α, ζ_1, ζ_2, ζ_3, and ζ are given in Table 1 for j=1, 2, 3.

We further choose Λ =200, $\mu_1 = 48$, $\mu_2 = 38$, and $\mu_3 = 25$. Also, let $f_{X_1}(t)$ and $f_{X_3}(t)$ be Erlang-2 distributions with $\nu_1 = \psi(d_1)\mu_1$ and $\nu_3 = \psi(d_3)\mu_3$ for cluster nodes 1 and 3 respectively, $f_{X_2}(t)$ is an Erlang-1 distribution with $\nu_2 = \psi(d_2)\mu_2$, where $\psi(d_j) = 1.5^{\log_2 d_j}$ for $j = 1, 2, 3$. Then, $\lambda = \Lambda/\alpha = 250$.

According to our algorithm in Section 4, we calculate the optimum numbers of d_1, d_2 and d_3 using the following steps.

We first solve for $\hat{a}^{(1)}$ in the Step a of Algorithm. That is, let us find the minimum value of $\hat{a}$ such that $F(t)|_{t=T^D} = F(T^D) \geq 0.98$, where $F(T^D)$ is computed by

$$F(t) = L^{-1}\Big\{\frac{200}{s(s+250)}\frac{\Pi_j^3 L(f_{X_j}(s))}{1-0.2\Pi_j^3 L(f_{X_j}(s))}\Big\}$$

and $L(f_{X_j}(t))$ is the LST of $f_{X_j}(t)$ for $j = 1, 2, 3$. Thus, we get $\hat{a} = 0.85$. Consequently, $d_1^{(1)} = 23$, $d_2^{(1)} = 34$, and $d_3^{(1)} = 68$.

Then, we use Step b of the Algorithm to compute $a_1^{(2)} = \max\{6.6774, 6.4780\}$ $= 6.6774$, $a_2^{(2)} = \max\{7.3099, 8.1828\} = 8.1828$ and $a_3^{(2)} = \max\{10.8696, 12.4379\}$ $= 12.4379$. Thus, $d_1^{(2)} = 26$, $d_2^{(2)} = 37$, and $d_3^{(2)} = 75$.

By using Step c,we get $d_1^M = 26$, $d_2^M = 37$, and $d_3^M = 75$. We further choose $d_j = d_j^M$, and verify that $I = \sum_{j=1}^3 c_j d_j = 729 < C^D$. This means that the optimum values are $d_1 = 26$, $d_2 = 37$ and $d_3 = 75$.

Extensive numerical results point to the fact that the proposed method provides an efficient way to calculate computing resources required for SLA assurance.

6 Conclusions

Cluster computing is excellent for parallel computation. It has become increasingly popular. We have proposed an approach for resource provisioning in a typical SLA-based cluster computing environment, whereby we minimize the total cost of computing resources allocated to a customer so that a given set of SLAs including percentile of the response time and cluster utilization is satisfied.

We have further formulated the resource provisioning problem as an optimization problem subject to SLA constraints for a typical SLA-based cluster computing system, and developed an efficient approach to solving the problem. Finally, we have demonstrated how to use our proposed approach to finding the

minimum values of computing resources required for the customer SLA guarantee by conducting numerical experiments.

Most importantly, we should point out that the proposed approach of this paper provides a framework for addressing and solving this type of resource provisioning problems subject to a given set of SLAs for high-performance computing systems including cluster and grid computing systems. Moreover, this approach can be extended to study a service request job model whose cluster nodes are arbitrarily linked as long as the defined link can be quantified. In this paper, we only considered a percentile of response time and cluster utilization in the SLA. Other metrics such as security, availability, vulnerability, and reliability will be discussed in another paper.

References

1. Aron, M., Sanders, D., Druschel, P., Zwaenepoel, W.: Scalable content-aware request distribution in cluster-based network servers. In: Proceedings of USENIX 2000 Technical Conference (June 2000)
2. Lucke, R.: Buidling Clustered Liux Systems. Prentice-Hall, Englewood Cliffs (2005)
3. Bucur, A.: Performance analysis of processor co-allocation in multicluster systems, PhD Thesis, Delft University of Technology, Delft, The Netherlands (2004)
4. Bucur, A., Epema, D.: Local versus global schedulers with processor co-allocation in multicluster systems. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2002. LNCS, vol. 2537, pp. 184–204. Springer, Heidelberg (2002)
5. Chang, J.: Processor performance: Update 1, http://SQL-Server-Performance.com
6. Cook, S.: The complexity of theorem proving procedures. In: Proceedings of the Third Annual ACM Symposium on the Theory of Computing, pp. 151–158. ACM, New York (1971)
7. Du, J., Leung, T.: Complexity of scheduling parallel task systems. SIAM Journal on Discrete Mathematics 2, 473–487 (1989)
8. Feitelson, D., Rudolph, L., Shwiegelshohn, U.: Parallel job scheduling: a status report. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2004. LNCS, vol. 3277, pp. 1–16. Springer, Heidelberg (2005)
9. Garey, M., Johnson, D.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman, New York (1979)
10. Heath, T., Diniz, B., Carrera, E.V., Meira Jr., W., Bianchini, R.: Self-configuring heterogeneous server clusters. In: Proceedings of the Workshop on Compilers and Operating Systems for Low Power (September 2003)
11. Jones, W.: Improving parallel job scheduling performance in multi-clusters through selective job co-allocation. PhD dissertation, Clemson University, Clemson, South Carolina, USA (2005)
12. Liu, C., Yang, L., Foster, I., Angulo, D.: Design and evaluation of a resource selection framework for grid applications. In: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing HPDC-11 2002 (HPDC 2002), Washington, DC, USA, p. 63. IEEE Computer Society, Los Alamitos (2002)

13. Levner, E.: Multiprocessor Scheduling: Theory and Applications. I-Tech Education and Publishing, Vienna (December 2007)
14. Ngubiri, J., Vliet, M.: Group-wise performance evaluation of processor co-allocation in multi-cluster systems. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2007. LNCS, vol. 4942, pp. 24–36. Springer, Heidelberg (2008)
15. Papadimitriou, C.: Computational Complexity, 1st edn. Addison-Wesley, Reading (1994)
16. Naik, V., Liu, C., Yang, L., Wagner, J.: On-line resource matching in a heterogeneous grid environment. In: Proceedings of the International Symposium on Cluster Computing and the Grid (CCGrid 2005). IEEE Computer Society, Los Alamitos (2005)
17. Shin, M., Chong, S., Rhee, I.: Dual-resource TCP/AQM for processing-constrained networks. In: Proceedings of the IEEE INFOCOM (April 2006)
18. Xiong, K.: Resource Optimization and Security in Distributed Computing. Ph.D. Dissertation, North Carolina State University, USA (December 2007)
19. Yom-Tov, E., Aridor, Y.: A self-optiimized job scheduler for heterogeneous server clusters. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2007. LNCS, vol. 4942, pp. 169–187. Springer, Heidelberg (2008)
20. Bouillet, E., Mitra, D., Ramakrishnan, K.: The structure and management of service level agreements in networks. IEEE Journal on Selected Areas in Communications 20(4), 691–699 (2002)
21. Chassot, C., Garcia, F., Auriol, G., Lozes, A., Lochin, E., Anelli, P.: Performance Analysis for an IP Differentiated Services Network. In: Proceedings of IEEE Internnational Conference on Communication (ICC 2002), pp. 976–980 (2002)
22. Cao, Z., Zegura, E.: Utility max-min: An application-oriented bandwidth allocaton scheme. In: Proceedings of the IEEE INFOCOM (March 1999)
23. Gaver, D., Handley, M., Padhye, J., Widmer, J.: Observing stochastic processes, and approximate transform inversion. Operation Research 14(3) (1966)
24. Graf, U.: Applied Laplace Transforms and z-Transforms for Scientists and Engineers. Birkhauser Verlag, Basel (2004)
25. INTERNAP, The INTERNAP route optimization solution: executive summary, http://www.internap.com/learning/whitepapers
26. Jacob, B., et al.: On Demand Operating Environment: Managing the Infrastructure, IBM Redbooks (June 2005)
27. Liao, R., Campbell, A.: Dynamic core provisioning for quantitative differentiated services. IEEE/ACM Transactions on Networking 12(3), 429–442 (2005)
28. Martin, J., Nilsson, A.: On service level agreements for IP networks. In: Proceedings of the IEEE INFOCOM (June 2002)
29. Menasce, D., Casalicchio, E.: A framework for resource allocation in grid computing. In: Proceedings of the MASCOTS (October 2004)
30. Paxson, V.: End-to-end Internet packet dynamics. In: Proceedings of the ACM SIGCOMM (1997)
31. Padhye, J., Firoiu, V., Towsley, D., Kurose, J.: Modeling TCP throughput: a simple model and its empirical validation. In: Proceedings of the ACM SIGCOMM (2004)
32. Perros, H.: Queueing Network with Blocking, Exact and Approximate Solutions. Oxford University Press, Oxford (1994)

33. Piessens, R.: Gaussian quadrature formulas for the numerical integration of Bromwich's integral and the inversion of the Laplace transform. Journal of Engineering Mathematics 5(1) (1971)
34. Stehfest, H.: Algorithm 386, numerical inversion of Laplace transforms. Communcations of the ACM 13(1) (January 1970)
35. Xiao, X., Ni, L.M.: Internet QoS: a big picture. IEEE Network (March/April 1999)
36. Zandt, T.: How to fit a response time distribution, http://citeseer.ist.psu.edu/552295.html

An Advance Reservation-Based Co-allocation Algorithm for Distributed Computers and Network Bandwidth on QoS-Guaranteed Grids

Atsuko Takefusa, Hidemoto Nakada, Tomohiro Kudoh, and Yoshio Tanaka

National Institute of Advanced Industrial Science and Technology (AIST)
{atsuko.takefusa,hide-nakada,t.kudoh,yoshio.tanaka}@aist.go.jp

Abstract. Co-allocation of performance-guaranteed computing and network resources provided by several administrative domains is one of the key issues for constructing a QoS-guaranteed Grid. We propose an advance reservation-based co-allocation algorithm for both computing and network resources on a QoS-guaranteed Grid, modeled as an integer programming (IP) problem. The goal of our algorithm is to create reservation plans satisfying user resource requirements as an on-line service. Also the algorithm takes co-allocation options for user and resource administrator issues into consideration. We evaluate the proposed algorithm with extensive simulation, in terms of both functionality and practicality. The results show: The algorithm enables efficient co-allocation of both computing and network resources provided by multiple domains, and can reflect reservation options for resource administrators issues as a first step. The calculation times needed for selecting resources using an IP solver are acceptable for an on-line service.

1 Introduction

Grid and network resource management technologies have enabled the construction of large-scale QoS-guaranteed Grid environments, which consist not only of performance-guaranteed multiple-computer clusters and storage resources, but also bandwidth-guaranteed networks linking the distributed resources. Several research projects have achieved coordination of resource managers for computers and network bandwidth and have constructed QoS-guaranteed Grid environments [1,2,3]. In contrast to canonical Grid environments, whose network resources are shared by abundant users, network links in these QoS-guaranteed Grids are dedicated to requesting users in order to guarantee the specified bandwidth.

In QoS-guaranteed Grid environments, each resource is managed by a local resource manager (RM) provided by several administrative domains or organizations, including commercial sectors. Therefore, each RM had better have an advance reservation capability, in order to provide a performance-guaranteed resource for a QoS-guaranteed Grid user, who also co-reserves other resources, including commercial resources. The KOALA [4] Grid scheduler and the QBETS [5]

E. Frachtenberg and U. Schwiegelshohn (Eds.): JSSPP 2010, LNCS 6253, pp. 16–34, 2010.

batch queue prediction service provide co-allocation of multiple cluster resources in coordination with RMs, without advance reservation, by acquiring and predicting the status of RMs. However, these strategies cannot guarantee to allocate the resources at the same time, so that the co-allocation user may have to pay for some resources, even if one resource may not be allocated at the expected time.

Therefore, "advance reservation" is one of the key technologies for a QoS-guaranteed Grid, and we have been working on development of the GridARS resource management framework[6] and the PluS plugin scheduler[7]. GridARS co-works with multiple RMs for computers, networks, and other resources, which manage the actual resources, and a reservation table of the resources, and co-allocates requested resources in advance for each QoS-guaranteed Grid user. PluS can be used in an RM and allows advance reservation on existing batch queuing systems, such as Sun GridEngine[8] and TORQUE[9], as well as Maui[10].

An important issue is then the question of Grid schedulers' advance reservation-based co-allocation of many kinds of distributed resources provided by various organizations. For building a QoS-guaranteed Grid, co-allocation algorithms have to select not only computers and storage resources, but also network links between the selected resources. Also, all of detailed resource allocation information in each RM will not be disclosed via commercial services. Grid schedulers for QoS-guaranteed Grids cannot apply either canonical Grid co-allocation algorithms [11,4,12] based on list-scheduling heuristic approaches, or network routing algorithms [13] based on Dijkstra's algorithm, straightforwardly. In addition, such a scheduling problem is known as NP-hard. It is important to determine co-allocation plans with short calculation time, especially for an on-line service.

Moreover, the co-allocation algorithms should reflect the following user and administrators scheduling options: In a user view, there should be options for resource co-allocation: (a) reservation time, (b) price, and (c) quality (availability). On the other hand, there should be options: (A) load balancing among RMs, (B) preference allocation to specific RMs because of energy savings or alliance issues, and (C) allocation suited for each user service level in an administrator view. Some studies[14,15,16,17] have already proposed advance reservation-based co-allocation algorithms for the both computer and network resources, but they have not adequately taken these options into account.

We propose an on-line advance reservation-based co-allocation algorithm for both computing and network resources on QoS-guaranteed Grids. The goal of our algorithm is to create reservation plans satisfying user resource requirements and to take the above co-allocation options in the user and administrator issues into consideration. In the proposed algorithm, our Grid scheduler (1) receives limited dynamic resource information from related RMs, (2) selects multiple combinations of suitable resources using the information, and (3) co-allocates the resources based on the selections. In phase (2), we modeled the co-allocation problem as an integer programming (IP) problem and applied IP solvers, in order to reflect the user and administrator options. We also describe how to apply the

options to these phases. This proposed algorithm could also be applied to co-allocation without advance reservation.

We evaluate the proposed algorithm in our advance reservation-based co-allocation model with extensive simulation, and show the validity of the algorithm in terms of functionality and practicality. Experiments on functional issues show that our algorithm enables efficient co-allocation of both computing and network resources provided by multiple domains, and can take administrator co-allocation options as a first step. In the experiments on practical issues, the calculation times of the proposed co-allocation method are acceptable for an on-line service.

2 Advance Reservation-Based Co-allocation Model

2.1 Overview of the Resource Management Framework

To enable a QoS-guaranteed Grid, we have been developing a Resource Management Framework called 'GridARS', as shown in Figure 1. Each of the domains, A and B, in Figure 1 denotes a network domain managed by a single administrative organization. This GridARS framework provides users with a QoS-guaranteed Grid, which spans several management domains, and is based on advance reservation.

The GridARS framework consists of a Global Resource Coordinator (GRC), which behaves as a Grid Scheduler, and Resource Managers (RMs), which manage each local resource. GRC and RM work together to provide users a QoS-guaranteed Grid. NRM, CRM, and SRM in Figure 1 denote Resource Managers for Networks, Computers, and Storages, respectively. More than one GRC is allowed in a single system. GRCs could be configured in a coordinated hierarchical manner, or in parallel, where several GRCs compete for resources with each other on behalf of their users. Some GRCs have a co-allocation planning capability, called Planner. Based on the reservation plans produced by a Planner, GRCs will perform resource reservation on subordinate GRCs or RMs.

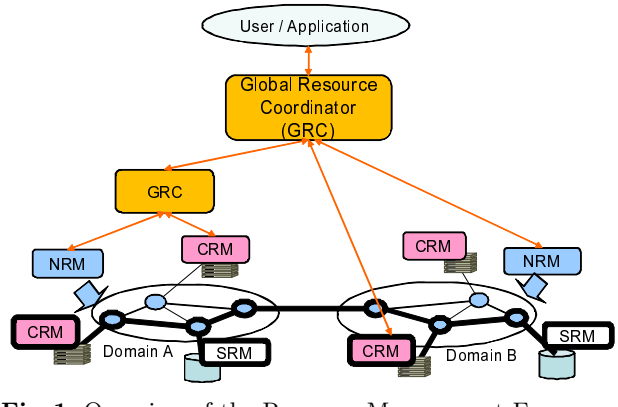


Fig. 1. Overview of the Resource Management Framework

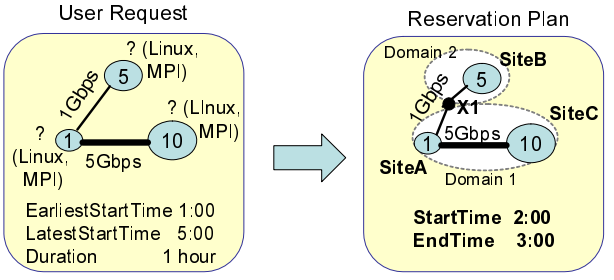


Fig. 2. User's resource request and a reservation plan from the Planner

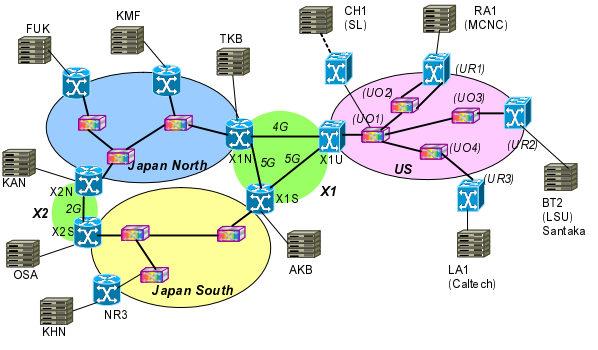


Fig. 3. Resources used for an experiment with G-lambda and EnLIGHTened

2.2 User Requests

We have performed several experiments on our QoS-guaranteed Grid, where we co-allocated several computing clusters, and light-path networks between those clusters, and ran real applications [1,2,18]. The applications include a molecular dynamics simulation program with GridRPC and MPI, and HD video live-streaming. The goal of the co-allocation algorithm proposed in this paper is to create resource reservation plans for simultaneous co-allocation.

On the left of Figure 2, we show a resource request from a user. On the right hand side, we show a plan generated by GRC Planner. For computing resources, users can specify the number of sites, the number of CPUs or cores for each site, and other attributes such as OSs. For network resources, users can specify bandwidth between the computing resource sites, latency, and other attributes, such as media types and availability. Users can also specify a time frame for each resource. In Figure 2, we specify EarliestStartTime (EST), LatestStartTime (LST), and Duration (D), where the user wants to reserve a time slot D units long, start after EST and before LST, i.e., finish before $LST + D$.

The GRC Planner gets a request from a user, selects resource groups from a resource pool, as shown in Figure 3, and creates reservation plans. Figure 3 shows a real resource pool used for an experiment performed by Japan's G-lambda project [19] and the United States' EnLIGHTened Computing project [20]. The

two projects achieved the world's first inter-domain coordination of resource managers for in-advance reservation of network bandwidth and compute resources between and among both the US and Japan in the fall of 2006 [2] D.

The reservation plan shown on the right hand side of Figure 2 demonstrates how computing resources (SiteA, SiteB, SiteC) and network resources between them are allocated and the start time and end time of the time slot are determined. Note that network topology produced by a Planner is not real network topology with real routers and switches, but an abstracted higher-level notation. This is because NRMs will be provided by the commercial sector, and abstract away the underlying real network configuration. In the planned topology, a network in a single domain is denoted as just a path. When the network spans several domains, it will be denoted as a set of paths connected together. In Figure 2, the network between SiteA and SiteC is denoted as a single path in Domain1, while the network between SiteA and SiteB is denoted as two paths in Domain1 and Domain2, connected at the domain exchange point X1.

2.3 Retrieving Available Resource Information from RMs

In order to have reservation planning, GRC has to retrieve available resource information for the future. In our co-allocation model, we assume that RMs will be provided by providers in the commercial sector, who will not disclose all the available resource information, including reservation time tables. The G-lambda project, which is a collaboration between industrial and governmental laboratories, AIST, KDDI R&D Laboratories, NTT, and NICT, has defined a web services-based resource reservation interface called GNS-WSI, which takes account of commercial services. GNS-WSI provides operations retrieving available resource information as well as reservation operations. We use the GNS-WSI retrieving operations, in which a requester has to specify a time frame to get available resource information.

3 An Advance Reservation-Based Co-allocation Algorithm

3.1 The Stages of Resource Co-allocation

We propose an on-line advance reservation-based co-allocation algorithm with the goal of creating reservation plans satisfying user resource requirements. The algorithm is invoked at every reservation request arrival. The stages of reservation planning and resource co-allocation in GRC are as follows:

1. GRC receives a co-allocation request from a user.
2. GRC Planner creates multiple reservation plans for the request.
   - 2i Planner selects N laddered time frames from $[EST, LST + D]$.
   - 2ii The Planner retrieves available resource information results at N time frames from RMs.

2iii Using this available resource information, the Planner determines N' ($N' \leq N$) reservation plans, based on a co-allocation method described in the next section.

2iv The Planner sorts N' plans by suitable order, which depends on co-allocation options in user and administrator issues.

3. In accordance with the reservation plans created by Planner, GRC tries to co-allocate the selected resources in cooperation with the subordinate RMs.
4. GRC returns co-allocation results, whether the resource co-allocation has succeeded or not, to the user. If it has failed, the user will resubmit a request with updated resource requirements.

As described in Section 2.2, a user can specify an exact reservation time or a range using the ESTCLSTC and D parameters. In the former case, GRC Planner creates reservation plans at the specified time frame. In the latter case, the Planner seeks available resources of available time frames in $[EST, LST+D]$. Therefore, the Planner creates multiple plans in stage 2. In stage 2i, it is possible to allow GRC administrators to make a trade-off between creating more suitable reservation plans with a large N and small planning cost with a small N. In stage 2ii, multiple query results are retrieved by a single query operation, using the GNS-WSI interface described in Section 2.3. In addition, GRC Planner can send queries to subordinate RMs concurrently. In stage 2iii, N reservation planning can running concurrently. Co-allocation options shown in stage 2iv are described in Section 3.4.

3.2 The Co-allocation Method Based on a General Optimization Problem

We propose a co-allocation method for both computing and network resources, modeled as an integer programming (IP) problem. This method is applied in stage 2iii.

Resource Notation. We denote resources as a directed graph $G = (V, E)$, as shown in Figure 4, where V is a set of vertices in G and E is a set of edges in G. Vertex v_q denotes a computing resource site or a network exchange point between network domains. Edges $e_{o,p}$ and $e_{p,o}$ denote network paths managed

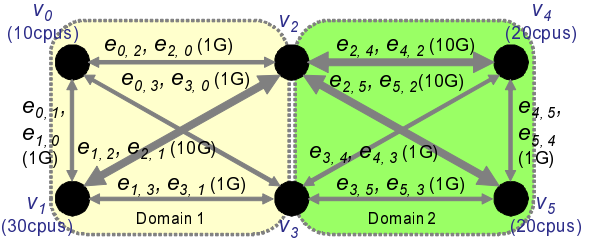


Fig. 4. Resources denoted as a graph

by NRMs. In Figure 4, there are two network domains (Domain1 and Domain2), which provide network paths. Here, $e_{o,p}$ denotes an edge from v_o to v_p, while $e_{p,o}$ denotes an edge from v_p to v_o. Parentheses attached to a vertex denote the number of available CPUs (or cores) at the sites, which will be referred to as $wc_i (i \in V)$. Parentheses attached to an edge denote the bandwidth of the path, which will be referred to as $wb_k (k \in E)$. Note that v_2 and v_3 in Figure 4 are network exchange points, which do not have any CPUs. We could add more attributes on vertices and edges, such as network latency or availability. The values per unit of each CPU and bandwidth are denoted as $vc_i (i \in V)$ and $vb_k (k \in E)$, respectively. These values will be prices or cumulative points to reflect co-allocation options. Note that wb_k and vb_k are shared by $e_{o,p}$ and $e_{p,o}$.

Resource Request Notation. Next, we denote a resource request from a user as a complete graph $G_r = (V_r, E_r)$, where V_r denotes required compute sites, and E_r denotes edges between V_r. The number of CPUs, provided by each compute site, and the network bandwidth are denoted as $rc_j (j \in V_r)$ and $rb_l (l \in E_r)$.

Modeling as a Mixed Integer Programming Problem. Now, we can plan resource reservation as the 0-1 integer programming (0-1 IP) problem to determine the following variables, with the parameters shown above. "0-1 IP" aims to find a combination of binary (0 or 1) variables to minimize or maximize an objective function subject to linear constraints.

$$x_{i,j} \in \{0,1\} \qquad (i \in V, j \in V_r) \tag{1}$$

$$y_{k,l} \in \{0,1\} \ (k = (m,n) \in E, m,n \in V, \\ l = (o,p) \in E_r, o,p \in V_r) \tag{2}$$

$x_{i,j}$ describes computing resource allocation, 1 means the requested resource denoted by the column is allocated to the actual resource denoted by the row. $y_{k,l}$ describes network resource allocation, 1 means the network path is taken, while 0 means it is not.

The objective function and constraints are described as follows:

Minimize

$$\sum_{i \in V, j \in V_r} vc_i \cdot rc_j \cdot x_{i,j} + \sum_{k \in E, l \in E_r} vb_k \cdot rb_l \cdot y_{k,l} \tag{3}$$

Subject to

$$\forall j \in V_r, \sum_{i \in V} x_{i,j} = 1 \tag{4}$$

$$\forall i \in V, \sum_{j \in V_r} x_{i,j} \leq 1 \tag{5}$$

$$\forall i \in V, \sum_{j \in V_r} rc_j \cdot x_{i,j} \leq wc_i \tag{6}$$

$$\forall l \in E_r, \sum_{k \in E} y_{k,l} \begin{cases} \geq 1 \ (rb_l \neq 0) \\ = 0 \ (rb_l = 0) \end{cases} \tag{7}$$

$$\forall k \in E, \sum_{l \in E_r} rb_l \cdot y_{k,l} \leq wb_k \tag{8}$$

$$\forall l = (o,p) \in E_r, \forall m \in V,$$

$$\sum_{n \in V, m \neq n} y_{(n,m),(o,p)} - \sum_{n \in V, m \neq n} y_{(m,n),(o,p)} = \begin{cases} x_{m,o} - x_{m,p} \ (rb_l > 0) \\ 0 \qquad\qquad\quad (rb_l = 0) \end{cases} \tag{9}$$

The objective function Equation (3) is meant to minimize the sum of the selected compute and network resources values.

Equations from (4) to (6) are constraints on computing resources, while Equations (7), (8) are constraints on network resources. Equation (9) is a constraint on both computing and network resources. Equation (4) ensures each compute site request j will be allocated on just one site. Equation (5) ensures each real site i will not be allocated to more than two sites. Equation (6) ensures each allocated site i has more CPUs than the required number. Equation (7) denotes that for a path l, the sum of $y_{k,l}$ is more than 1 when a user requests bandwidth on path l, and 0 when a user does not. The sum will become 1 if the path is included in a single domain, and become n if the path spans n domains. Equation (8) denotes real path k can provide more bandwidth than required.

Equation (9) is derived from the mass balance constraints[21], which claim that at any vertex on a graph, total inflows plus generation on the vertex are equal to total outflows. Assume a path of flow f with one intermediate node between start and end. Here, generations are f from the start point, $-f$ from the end point, and 0 from the intermediate node of the path. Assume we have a bandwidth reservation request for path l. From application of the mass balance constraint with flow $f = 1$, for each path $l = (o, p)io$ denotes a start point and p denotes an end point of l) and each m (a computing resource site or a network exchange point), we obtain Equation (9). The value of Equation (9) will be 1 if m is the start point, and -1 if m is the end point, and 0 if m is the others. Here, $x_{m,o} = 1$ when m is the start point, $x_{m,p} = 1$ when m is the end point, and $x_{m,o} = x_{m,p} = 0$ when m is neither the start nor end point. Therefore, the right of Equation (9) could be represented as $x_{m,o} - x_{m,p}$. Thus, this equation ties $x_{i,j}$ and $y_{k,l}$ together.

The proposed co-allocation method, based on a general optimization problem, could be applied to co-allocation without advance reservation. It is also effective when some of the resources are specified by the users in advance.

3.3 Additional Constraints for Optimization

Generally, calculation times of general optimization problems, including 0-1 IP, become exponentially long when the number of variables becomes large, due

to NP-hard. We propose additional constraints, which are expected to make calculation times of our co-allocation method shorter.

Subject to

$$\forall l \in E_r, \forall m, n \in V (m \neq n),\ y_{(m,n),l} + y_{(n,m),l} \leq 1 \tag{10}$$

$$\forall l \in E_r,\ \sum_{k \in E} y_{k,l} \leq P_{max} \tag{11}$$

Equation (10) indicates that both of the directed edges between the same two points, (m, n) and (n, m), are not selected in each requested network. Equation (10) enables solvers to avoid redundant search for an optimal solution. Equation (11) specifies P_{max}, the maximum number of paths, which make up each requested network. Here, P_{max}, given heuristically, makes the search area smaller and calculation time prospects shorter, although we might not be able to find an optimal solution, whose network consists of more than P_{max} paths.

3.4 Reflecting Co-allocation Options in the Algorithm

As mentioned in Section 1, there are co-allocation options in user and GRC administrator issues: A user uses her co-allocation option to prioritize (a) reservation time, (b) price, and (c) quality (availability), in addition to general resource requirements. A GRC administrator has options to prioritize (A) load balancing among RMs, (B) preference allocation to specific RMs, and (C) allocation suited for each user service level.

These co-allocation options can be reflected in the proposed algorithm as follows: For option (a), we sort reservation plans by late reservation time in stage 2iv. For (b), we set the values vc_i and vb_k to CPU and bandwidth unit prices and sort plans by the total price in stage 2iv. For (c), we set vc_i and vb_k to their points, such as levels of fault tolerance, and sort plans by the total points in stage 2iv. To fulfill the administrator's options (A) and (B), we have to weight each resource and add other objective functions. Option (C) could be handled by modification of the available resource retrieval information, which reflects service level requirements from the users.

4 Experiments

4.1 Simulation Model

We conduct simulations to investigate the validity of our co-allocation algorithms, in terms of functionality and practicality. In the experiments on the functional issues, we investigate if the algorithm can schedule both computing and network resources from multiple domains efficiently, and if our algorithm can take co-allocation options in user and administrator issues into consideration. In the experiments on the practical issues, we compare the calculation

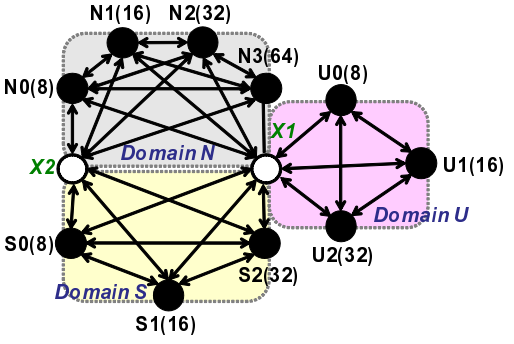


Fig. 5. Simulation environment

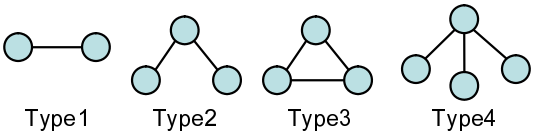


Fig. 6. Resource requirement types

times of our algorithm with/without additional constraints, Equation (10) and Equation (11), applying different IP solvers.

In the both simulations, we assume the experimental environment shown in Figure 3, used in the EnLIGHTened and G-lambda (ELGL) experiments. The environment consists of three network domains and two domain exchange points, as shown in Figure 5. In-domain computing resource sites, denoted by black circles, are inter-connected by a complete graph and each domain exchange point, denoted by a white circle, and each in-domain site is connected to every other, respectively.

An overview of simulation settings is given in Table 1. In our simulations, there are two users, UserA and UserB, and each user requests resource co-allocation, repeatedly, as shown in Figure 6. Each user request arrives in the first 24 hours and it reserves resources for the next 24 hours. Interarrival rate of each user request is set to 407.327 [sec], so that the request loads are set to 10 [%] at 144 [min] to 100 [%] at 1440 [min]. The number of reservation plans N in the GRC Planner is set to 10. For each request, co-allocation plans are sorted by reservation time, and applied the (a) reservation time option. In the experiments, we assume a smallish numbers of CPUs at each site (8 - 64 CPUs) and the requested site (1-8), because the calculation times of our algorithm does not depend on the number of CPUs, but on the number of sites.

4.2 Experiments on Functional Issues

First, we compare success ratios of resource co-allocation among two users, UserA and UserB and investigate the functionality needed to reflect the administra-

Table 1. Simulation settings

Simulation environment settings	
Configuration	No. of GRC=1, No. of NRM=3 (N, S, U), No. of CRM=10
No. of sites / domain name	4/N, 3/S, 3/U
Domain exchange points	X1 (N, S, U), X2 (N, S)
No. of CPUs	N{8, 16, 32, 64}, S{8, 16, 32}, U{8, 16, 32}
CPU unit value	1
Bandwidth [Gbps]	in-domain paths : 5, inter domain paths : 10
Bandwidth unit values	in-domain paths : 5, inter domain paths : 3
Resource requirement settings	
Users	UserA, UserB
Resource requirement types	Type1,2,3,4 (Uniform distribution)
Requested No. of CPUs	1, 2, 4, 8 for all sites in Type1,2,3,4 (Uniform distribution)
Requested bandwidth [Gbps]	1 for all paths in Type1,2,3,4 (Fixed)
Interval of each user request	Poisson arrivals
Reservation duration [min]	30, 60, 120 (Uniform distribution)
LST - EST	Reservation duration × 3

tor option (C). In this experiment, we used GLPK (GNU Linear Programming Kit) [22] as a solver for 0-1 IP in the proposed algorithm. We assume that the users co-allocation option is (a), and the administrator options are (A) and (C). In the experiments with option (C), the service level (SL) of UserB is set to low: UserB can book half of the available resources, while UserA can book all of them.

Figure 7 shows success ratios of resource co-allocation, requested by UserA and UserB, respectively. The horizontal axis indicates elapsed time in each simulation

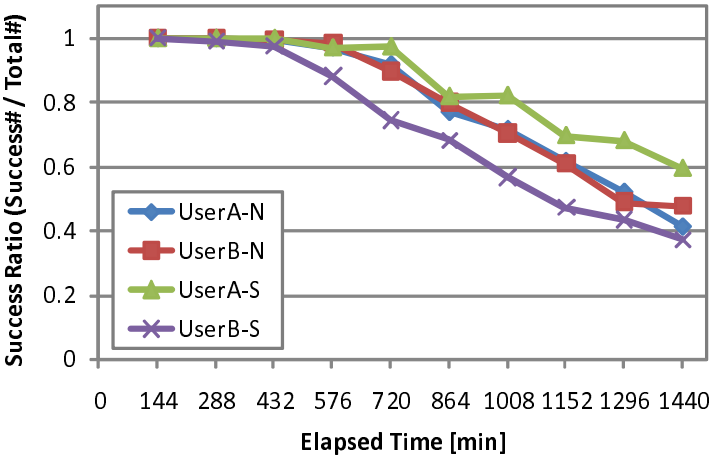


Fig. 7. Comparison of resource co-allocation success ratios. The request load varies from 10 [%] to 100 [%].

and the vertical axis indicates the success ratio. Each plot shows the average success ratio of requests that arrived between 0 and 144 [min] to between 1296 and 1440 [min] in 10 simulation runs, respectively. The request load is 0-10 [%] between 0 and 144 [min] and 90-100 [%] between 1296 and 1440 [min]. "UserA" and "UserB" denote UserA and UserB, and "-N" and "-S" denote results with option (A) and (C) applied. UserB is set to a low SL.

The results of a normal case ("-N") show that success ratios of UserA and UserB are 0.918 and 0.897, when the request load = 50 [%] (720 [min]), and still 0.618 and 0.609, when the load = 80 [%] (1152 [min]). The main result here is that the proposed algorithm is effective for co-allocation of multiple computing and network resources spanning over multiple network domains.

In comparison of service levels, success ratios of UserA and UserB are comparable in the results with option (A) ("-N") applied. On the other hand, UserA's results show better success ratios, 0.595 when request load = 100 [%], than UserB's results, 0.374, in the option (C) results. Therefore, Figure 7 shows that the algorithm can take option (C) into consideration.

Next, we compare the co-allocation results with administrator options (A) and (B). In the cases with option (B), specific sites are prioritized by the weights of CPUs. Figure 8 shows the results of applying option (A) (top), option (B) prioritized by the number of CPUs in each site (middle), and option (B) prioritized by network domains (bottom). Each CPU unit value is set to 1 in the top cases, 1, 10, 100, 1000 for 64, 32, 16, 8 CPU sites in the middle cases, and 1, 10, 100 for domain N, S, U sites in the bottom cases. Our algorithm selects resources to minimize total resource weight.

The simulation results show that resource utilization of the top graph increases almost uniformly. On the other hand, sites with many CPUs and sites belonging to domain N are preferentially selected in the middle and bottom graphs if the total load of resource requests is not high. Therefore, the experimental results prove that our algorithm can takes co-allocation options in administrator issues into consideration.

4.3 Experiments on Practical Issues

The goal of our algorithm is to be used in an on-line service. However, our algorithm is modeled as 0-1 IP, and so its calculation time becomes drastically long when the number of valuables becomes large, due to NP-hard. Therefore, we confirm our algorithm is practical when used to compare the calculation times of our algorithms with/without additional constraints in Section 3.3, applying different IP solvers.

In the comparison of IP solvers, we apply free open source solvers, GLPK (GNU Linear Programming Kit)[22] and a satisfiability problem (SAT) based solver, Sugar++[23] with a SAT solver, MiniSat[24]. Sugar++ enables a SAT solver to solve an optimization problem, which maximizes or minimizes its objective functions. Sugar++ temporally determines the maximum or minimum value of the objective function and solves a SAT problem using the SAT solver, repeatedly. Then, Sugar++ finds an optimal solution.

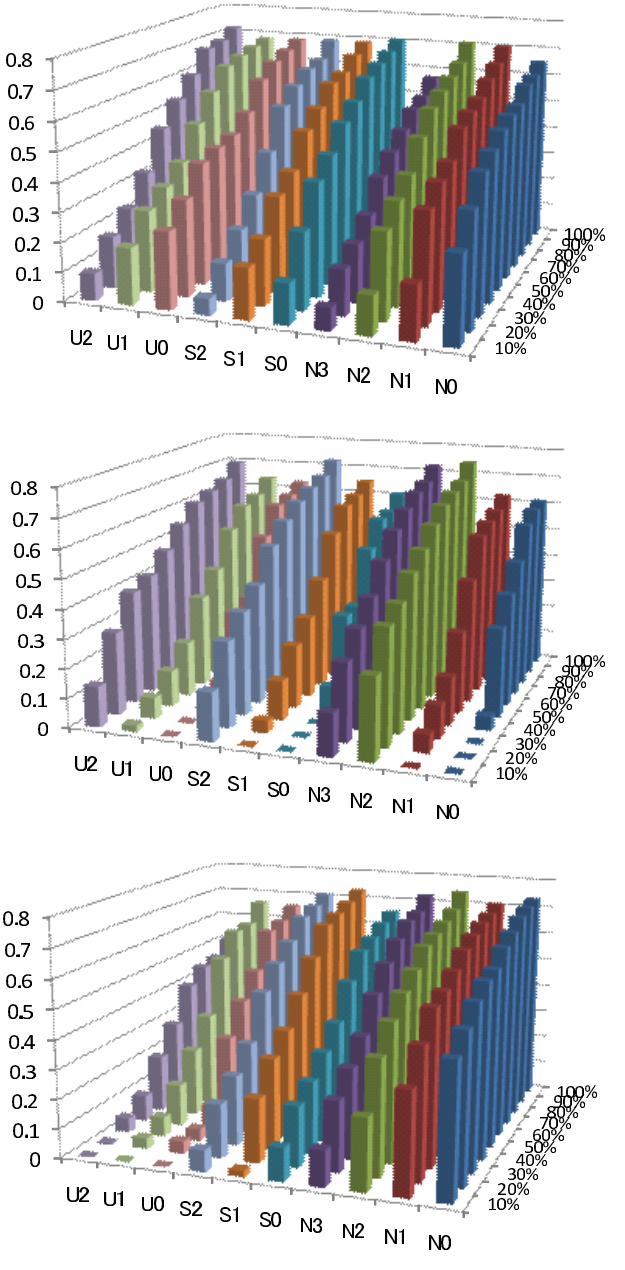


Fig. 8. Comparison of resource utilization between sites. Three patterns of administrator's options are applied, option (A) (top), option (B) prioritized by the number of CPUs in each site (middle), and option (B) prioritized by network domains (bottom). For each axis, N0 - U2 indicate compute resource site names, 10 % - 100 % indicate request load, and 0 - 0.8 indicate resource utilization of each site, respectively.

Table 2. Comparison of calculation times of 0-1 IP

	Avg [sec]	Max [sec]	σ
GLPK	0.779	8.492	1.721
GLPK-st	0.333	4.205	0.700
MiniSat-st	12.848	216.434	27.914
MiniSat-st-1	1.918	2.753	0.420

Experimental Results of Calculation Times. We compare four patterns of solvers and constraints as follows:

GLPK: GLPK is applied.
GLPK-st: GLPK with additional constraints in Section 3.3 is applied.
MiniSat-st: Sugar++ and MiniSat with the additional constraints are applied.
MiniSat-st-1: Sugar++ and MiniSat with the additional constraints are applied, however, this solves a SAT problem once only, and does not obtain an optimal solution.

For each -st version, P_{max} is set to 2 in the experiments.

Table 2 shows the average, maximum, and standard deviation (σ) of calculation values after applying the different combinations shown above. The results of GLPK and GLPK-st show that GLPK-st is twice as fast than GLPK without additional constraints. From the results one can see that much improvement can be gained by applying additional constraints for IP problems. In our comparison of solvers, IP-based GLPK-st and SAT-based MiniSat-st, the GLPK-st results show much shorter times than the MiniSat-st ones. The results here indicate IP-based solvers are quite suitable for our scheduling problems. However, MiniSat-st-1 shows the best performance of all combinations, in terms of the maximum values and standard deviations.

Next, we wish to compare the average calculation times for each request in Figure 9 and Figure 10. The horizontal axis denotes the request number and each plot is the average calculation time of $N = 10$ reservation plans for each request, because these N plans can be solved independently in the embarrassingly parallel (EP) manner. The vertical axis of Figure 9 denotes elapsed time in log scale and the results from 0 to 10 [sec] are shown in Figure 10.

Figure 9 indicates that the calculation times of solvers needed to obtain an optimal solution are rather dispersed, while those of MiniSat-st-1 are not. However, the dispersion decreases when the request number becomes large. The main results here indicate that the search areas of IP problems become small and the calculation times decrease, when the available resources decrease.

In Figure 10, one can see the three lines of the results of applying MiniSat-st-1. Therefore, the calculation time of MiniSat-st-1, which satisfies all of the constraints, but does not obtain an optimal solution, is proportional to the number of vertices in Figure 6.

Discussion. There are lots of scheduling studies applying a sort of heuristic method, because scheduling problems are known as NP-hard. On the other hand,

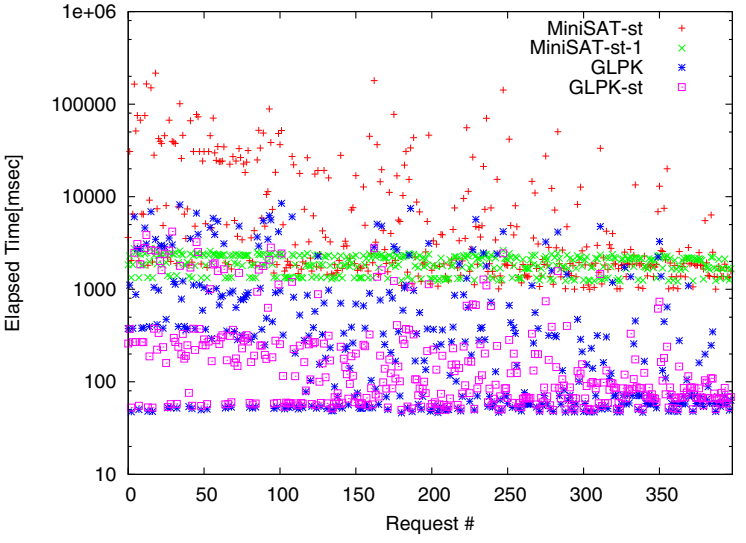


Fig. 9. Comparison of the average calculation times for each resource request. The vertical axis indicates elapsed time in log scale.

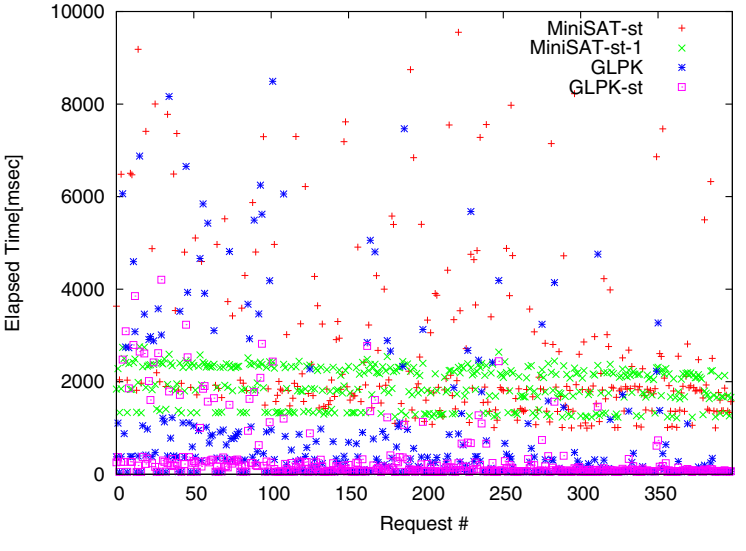


Fig. 10. Comparison of the average calculation times for each resource request. The graph shows the results from 0 to 10 [sec].

the coverage of IP problems is expanding, because of the recent rapid increase in computer performance and the improvement of IP algorithms and solvers. Also, the IP calculation times can be reduced by applying additional constraints and

approximate solutions, which does not obtain an optimal solution. Commercial IP solvers, such as ILOG CPLEX[25], are also known to reduce calculation times by applying additional constraints in pre-processing. In addition, approximate solutions can provide a solution, which is not optimal, but close to an optimal solution, with a short calculation time. Therefore, approximate solutions seem efficient for scheduling problems, which do not need an optimal solution.

While the number of variables in the proposed algorithm becomes N^3 depending on the number of computer sites N, our co-allocation problem has the following characteristics and modeling as an IP problem is an effective approach for the problem:

- The search area of a single GRC can be localized, because GRCs are located hierarchically as shown in Figure 1.
- The number of variables scales by the number of computer sites, not computers.
- In practical use, additional constraints will be defined, such as those for communication latencies, resource hardware requirements, and execution environments.

5 Related Work

There are several Grid scheduling algorithms for both computing and network resources. The differences between our algorithms are described as follows:

The VIOLA project has work on development of the MetaScheduling Service (MSS)[14], which co-allocates both computing and network resources, based on advance reservation. Roblitz proposed a Grid scheduling algorithm of co-reservation for multiple resources, based on general optimization problems[15]. The differences between the above two algorithms and ours are: their GRC can obtain all of the reservation time tables managed by local resource managers and their algorithms assume a simple network resource model, such as a single domain and single switch configuration.

Ando and Aida proposed a Grid scheduling algorithm for both computing and network resources, and modeled a single domain network and multiple switch configuration[16]. Their algorithm, based on a general backtrack approach, reserves computers and the related network paths incrementally, releases the reserved resources when the next required resource could not be discovered, and then finds the next candidate. This results in a complicated co-allocation process and blocking of many resources during the process.

Elmroth and Tordsson also propose a co-allocation algorithm[17] for NorduGrid[26]. Their algorithm fixes a reservation time and searches a combination of required computers first, and the related network paths next. If it cannot find the requested resources, it slides the reservation time frame and searches for resources in the same manner, repeatedly. This approach causes a long planning time, when constraints of the latter resources are strict, such as less network bandwidth available, and when resource co-allocation is failed, while our approach does not.

Both backtrack and NorduGrid approaches were not able to find suitable resources, e.g., due to expensive price, long communication latency, and redundant path networks, because they select the first found resources. On the other hand, our algorithm can take co-allocation options in user and administrator issues into consideration and find suitable resources.

Netto and Buyya proposed automatic rescheduling of multiple co-allocation requests of computing resources based on advance reservation[27], in order to achieve high utilization. However, it is difficult to reschedule various allocated resources automatically, when the number of allocated resources, including networks, becomes larger and the resources are provided by commercial entities, which do not disclose the detailed reservation time tables and also charge for the resources.

Rescheduling is an important issue for a QoS-guaranteed Grid not only to increase system utilization but to recover failures. Our approach is that our monitoring system[28] provides a user monitoring information on the reserved resources and the user can send a modification request to our co-allocation system, if required. The proposed algorithm can be applied to such a modification request.

6 Conclusions and Future Work

We propose an on-line advance reservation-based co-allocation algorithm for both computing and network resources on QoS-guaranteed Grids, constructed over multiple network domains. The proposed IP-based algorithm can create reservation plans satisfying user resource requirements and takes co-allocation options in user and administrator issues into consideration. The proposed algorithm could also be applied to co-allocation without advance reservation.

Our experimental results showed the validity of the proposed algorithm, in terms of both functionality and practicality: Our algorithm enables efficient co-allocation of both computing and network resources provided by multiple domains, and can reflect reservation options in administrator issues. The calculation time needed for selecting resources is acceptable for an on-line service.

For future work, we will improve our algorithm and conduct further experiments on the scalability with more actual constraints, such as communication latencies, resource hardware requirements, and execution environments. We also plan to apply sophisticated economy models for resource pricing and SLA models on resource provider sides, and will confirm that our algorithm can also take user co-allocation options efficiently under these situations.

Acknowledgments

We would like to thank Prof. Naoyuki Tamura and Mr. Tomoya Tanjo from Kobe University, and Prof. Katsuki Fujisawa and Mr. Yuichiro Yasui from Chuo University for their generous support with respect to optimization problems.

This work was partly funded by KAKENHI 21700047 and the National Institute of Information and Communications Technology.

References

1. Takefusa, A., Hayashi, M., Nagatsu, N., Nakada, H., Kudoh, T., Miyamoto, T., Otani, T., Tanaka, H., Suzuki, M., Sameshima, Y., Imajuku, W., Jinno, M., Takigawa, Y., Okamoto, S., Tanaka, Y., Sekiguchi, S.: G-lambda: Coordination of a Grid Scheduler and Lambda Path Service over GMPLS. Future Generation Computing Systems 22(2006), 868–875 (2006)
2. Thorpe, S.R., Battestilli, L., Karmous-Edwards, G., Hutanu, A., MacLaren, J., Mambretti, J., Moore, J.H., Sundar, K.S., Xin, Y., Takefusa, A., Hayashi, M., Hirano, A., Okamoto, S., Kudoh, T., Miyamoto, T., Tsukishima, Y., Otani, T., Nakada, H., Tanaka, H., Taniguchi, A., Sameshima, Y., Jinno, M.: G-lambda and EnLIGHTened: Wrapped In Middleware Co-allocating Compute and Network Resources Across Japan and the US. In: Proc. GridNets 2007 (2007)
3. AAA scenarios and test-bed experiences. Deliverable reference number:d4.2, The PHOSPHORUS project (2008), http://www.ist-phosphorus.eu/files/deliverables/Phosphorus-deliverable-D4.2.pdf
4. Mohamed, H., Epema, D.: Experiences with the KOALA Co-Allocating Scheduler in Multiclusters. In: Proc. 5th IEEE/ACM Int'l Symp. on Cluster Computing and the GRID (CCGrid 2005) (May 2005)
5. Nurmi, D., Brevik, J., Wolski, R.: QBETS: Queue Bounds Estimation from Time Series. In: Proc. 13th Workshop on Job Scheduling Strategies for Parallel Processing (2007)
6. Takefusa, A., Nakada, H., Kudoh, T., Tanaka, Y., Sekiguchi, S.: GridARS: An Advance Reservation-based Grid Co-allocation Framework for Distributed Computing and Network Resources. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2007. LNCS, vol. 4942, pp. 152–168. Springer, Heidelberg (2008)
7. Nakada, H., Takefusa, A., Ookubo, K., Kishimoto, M., Kudoh, T., Tanaka, Y., Sekiguchi, S.: Design and Implementation of a Local Scheduling System with Advance Reservation for Co-allocation on the Grid. In: Proceedings of CIT 2006 (2006)
8. Sun Grid Engine, http://gridengine.sunsource.net/
9. TORQUE Resource Manager, http://www.clusterresources.com/resource-manager.php
10. Maui Cluster Scheduler, http://www.clusterresources.com/pages/products/maui-cluster-scheduler.php
11. Czajkowski, K., Foster, I., Kesselman, C.: Resource co-allocation in computational grids. In: Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8), pp. 219–228. IEEE Computer Society, Los Alamitos (1999)
12. Castillo, C., Rouskas, G.N., Harfoush, K.: Resource Co-Allocation for Large-Scale Distributed Environments. In: Proc. HPDC 2009, pp. 137–150 (2009)
13. Taesombut, N., Chien, A.A.: Evaluating Network Information Models on Resource Efficiency and Application Performance in Lambda-Grids. In: Proc. SC 2007 (November 2007)
14. Barz, C., Pilz, M., Eickermann, T., Kirtchakova, L., Waldrich, O., Ziegler, W.: Co-Allocation of Compute and Network Resources in the VIOLA Testbed. TR 0051, CoreGrid (September 2006)
15. Roblitz, T.: Global Optimization For Scheduling Multiple Co-Reservations In The Grid. In: Proc. CoreGRID Symposium, pp. 93–109 (August 2008)

16. Ando, S., Aida, K.: Evaluation of Scheduling Algorithms for Advance Reservations. In: IPSJ SIG Notes 2007-HPC-113, pp. 37–42 (2007)
17. Elmroth, E., Tordsson, J.: A standards-based Grid resource brokering service supporting advance reservations, coallocation and cross-Grid interoperability. Concurrency and Computation: Practice and Experience 25(18), 2298–2335 (2009)
18. Kudoh, T.: GRID computing and a role of photonic networks. In: Proc. SPIE Asia Pacific Optical Communications (APOC 2008) 7137, 713713 (2008)
19. The G-lambda project, http://www.g-lambda.net/
20. The EnLIGHTened Computing project, http://enlightenedcomputing.org/
21. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows: Theory, Algorithms, and Applications. Prentice Hall, Englewood Cliffs (1993)
22. GLPK (GNU Linear Programming Kit), http://www.gnu.org/software/glpk/glpk.html
23. Tanjo, T., Tamura, N., Banbara, M.: Sugar++: A SAT-based Max-CSP/COP Solver. In: Proc. the Third International CSP Solver Competition, pp. 144–151 (2008)
24. MiniSat, http://minisat.se/
25. ILOG CPLEX, http://www.ilog.co.jp/product/opti/cplex/cplex.html
26. NorduGrid, http://www.nordugrid.org/
27. Netto, M.A.S., Buyya, R.: Rescheduling Co-Allocation Requests based on Flexible Advance Reservations and Processor Remapping. In: Proc. Grid 2008, pp. 144–151 (2008)
28. Takefusa, A., Nakada, H., Yanagita, S., Okazaki, F., Kudoh, T., Tanaka, Y.: Design of a Domain Authorization-based Hierarchical Distributed Resource Monitoring System in cooperation with Resource Reservation. In: Proc. HPC Asia 2009, pp. 77–84 (September 2009)

A Greedy Double Auction Mechanism for Grid Resource Allocation

Ding Ding, Siwei Luo, and Zhan Gao

Beijing Jiaotong University, Beijing 100044, China
{dding,swluo}@bjtu.edu.cn,
bjtugaozhan@gmail.com

Abstract. To improve the resource utilization and satisfy more users, a Greedy Double Auction Mechanism(GDAM) is proposed to allocate resources in grid environments. GDAM trades resources at discriminatory price instead of uniform price, reflecting the variance in requirements for profits and quantities. Moreover, GDAM applies different auction rules to different cases, over-demand, over-supply and equilibrium of demand and supply. As a new mechanism for grid resource allocation, GDAM is proved to be strategy-proof, economically efficient, weakly budget-balanced and individual rational. Simulation results also confirm that GDAM outperforms the traditional one on both the total trade amount and the user satisfaction percentage, specially as more users are involved in the auction market.

1 Introduction

Due to the specialities such as geographical distribution, heterogeneity and site autonomy, it is hard and challenging to manage grid resources. Fortunately, with the emergence of the grid economy, economic-based model is proposed to allocate resources in grid. This economic mechanism includes auctions, commodity markets, tenders and posted price and has many attractive features[1,2].

Auction-based resource allocation has attracted much attentions since it requires less global information, has decentralized structure and is easy to implement. Depending on the type of interactions between sellers and buyers, auctions can be classified into two classes, one-sided auctions and two-sided auctions. In one-sided auctions, only grid users submit bids to a central auctioneer, while in two-sided auctions, also called double auctions, both users and resource owners submit bids. The selling price and the users and the resource owners that trade are decided by the central auctioneer based on different types of double auction mechanisms. According to the trading units, the double auction can be classified as SDA (Single-unit Double Auction), where at most one unit of resource can be traded in one auction, and MDA (Multi-unit Double Auction), where more than one unit of resource can be traded in one auction. MDA is more suitable for a huge number of buyers and sellers trading through network, thus can be well applied to resource allocation in a grid environment.

E. Frachtenberg and U. Schwiegelshohn (Eds.): JSSPP 2010, LNCS 6253, pp. 35–50, 2010.

The earliest work on economic-based resource allocation can be traced back to 1968 when Sutherland proposed the auction mechanism for resource allocation in PDP-1 machine[3]. GRACE [4] presented a economic-based grid resource management architecture and described several economic models. Popcorn [5] was a Web based computing market, which adopted both one-sided and two-sided auctions to realize on-line resource allocation across the Internet. Spawn [6] managed heterogeneous computer resources based on Vickery auction, so did CORE [7]. Double auctions were used in JaWS [8] and preferred by many projects such as Tycoon [9]. However, these projects only adopted the basic auction methods instead of researching them deeply. In [10] and [11], a SDA market is designed, but only the strategy behavior on the buyers side was considered. Still in a SDA, [12] considered strategy behaviors both on the sellers and buyers sides. In [13], a MDA mechanism was proposed for the electric market, using the uniform auction. Different double auction protocols are compared in [14] and combinatorial MDA for resource allocation was studied in [15] and [16]. Double auction is also further used in grid scheduling[17,18].

In this paper, we propose a double auction based resource allocation mechanism, a greedy MDA, trying to make the resource consumers and providers trade as more as possible under the guarantee of their QoS demands(that is the requirements for prices and quantities).

The rest of this paper is organized as follows. The next section gives the basic framework of the auction market. In section 3 the disadvantages of the traditional auction mechanism are described and GDAM is presented. Some features of GDAM are proved in section 4. In section 5, simulation experiments are conducted and the results are discussed. The last section includes the conclusion of this paper.

2 The Basic Market Framework

Generally, entities that can be traded in a grid market are different kinds of grid resources, such as storage resource and computing resource. In this paper we focus on auctions for only one kind of grid resource and the combinatorial auction is not considered. The auction market is constituted by three components, Seller Agent (SA), who works on behalf of sellers, Buyer Agent (BA), who works on behalf of buyers and Auction Agent (AA), who manages the auction market. A buyer will send to its BA the buy request, $br = (q, p)$, where $br.q$ is the quantity of the resource it needs and $br.p$ is the bidding price for a unit of the resource. A seller will send to its SA the sell request, $sr = (q, p)$, where $sr.q$ is the quantity of the resource it wants to sell and $sr.p$ is the asking price for a unit of the resource. And neither a SA nor a BA knows the treading requests of other agents. After receiving the trading requests from SAs and BAs, AA makes use of a certain double auction mechanism to decide the buyers and seller who can trade and the amounts and prices of the resources that will be traded. AA organizes double auctions at regular time interval, and BAs and SAs submit their own trading requests at the auction period.

The market framework is illustrated in figure 1.

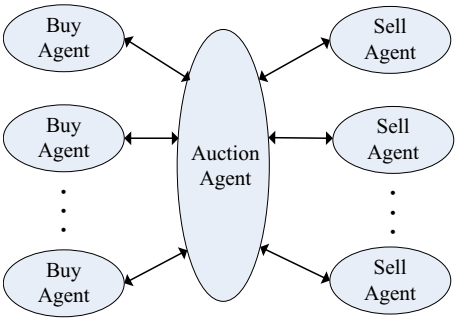


Fig. 1. The basic market framework

3 Greedy Double Auction Mechanism

Most researches on double auction in the economic field aim at maximizing the total market value, that is, maximizing the collective surplus of the market participants. However, these mechanisms are not quite suitable for the grid. Firstly, before the final trading price is determined by AA, neither SAs nor BAs know how much profit they will get from trading per unit of resource. Thus, the market can not guarantee a definite profit for the participants. Secondly, under traditional mechanisms the final trading price for all the grid resources is the same, which is unfair. A resource with a better quality may have a higher cost, and is supposed to be traded at a higher price. Thirdly, in order to maximize the collective surplus, traditional mechanisms tend to trade the buyers and sellers with the largest gaps between their reservation prices. This causes two problems. On one hand, the market value is shared by the minority and is distributed unbalanced for only a small percentage of participants can make successful trade, especially when the number of participants is large. On the other hand, the utilization of grid resources is low. Few buyers can eventually benefit from the resources they need, making many resources, which can potentially be traded and utilized, idle. In this paper, a greedy double auction mechanism(GDAM) is proposed which does not focus on how to maximize the total market value but how to improve the resources utilization and benefit the majority of the market participants.

Under GDAM, the auction market provides trading service for the participants, using trading price and trading amount as two QoS parameters. As for a seller j he requires the selling service with $sr_j.p$ and $sr_j.q$ as his QoS requirements; as for a buyer i he requires the buying service with $br_i.p$ and $br_i.q$ as his QoS requirements. A seller's asking price is his expected profit plus the cost of providing per unit of resource, while a buyer gets his bidding price by subtracting the cost from the value created by consuming per unit of the resource.

Let seller j's cost of providing per unit of resource be sc_j and the value created by buyer i's consuming per unit of resource be cv_i. Let q_{ij} denote the quantity buyer i buys from seller j, then the surplus of buyer i for this transaction is

$$sb_i = (cv_i - br_i.p)br_i.q. \tag{1}$$

and the surplus of seller j is

$$ss_j = (sr_j.p - sc_j)sr_j.q. \tag{2}$$

By dong this, the profits of the successful participants can be guaranteed. With trading at the expected price, no agents will complain about the unfairness. And our mechanism works in three different cases.

Case I: Supply over Demand (SoD). In this case, there are more supply quantities than demand quantities available in the auction market. Let m be the number of buyers and n be the number of sellers then (3) holds.

$$\sum_{j=1}^{n} sr_j.q \geq \sum_{i=1}^{m} br_i.q + T \ (T > 0). \tag{3}$$

Without loss of generality, we assume

$$br_1.p \geq br_2.p... \geq br_m.p. \tag{4}$$

$$sr_1.p \geq sr_2.p... \geq sr_n.p. \tag{5}$$

According to (4) and (5), GDAM arranges the demand quantities and supply quantities in the descending order of price(refer to figure 2). We can see from figure 2 that the price-quantity broken line of the buyers crosses that of the sellers at some crossing points. These crossing points can be divided into two

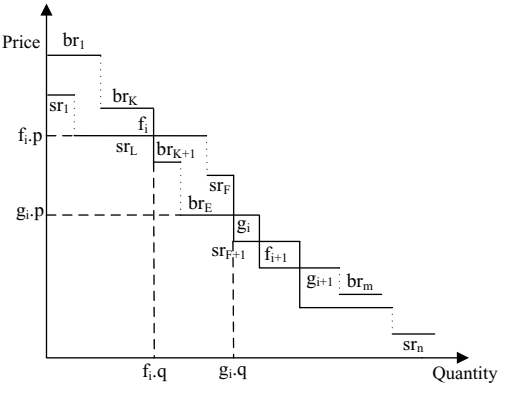


Fig. 2. Supply over Demand

classes: up-crossing point f_i and down-crossing point g_i. f_i occurs when (6) and (7) hold, and we call br_K and sr_L up critical requests.

$$br_{K+1}.p < sr_L.p < br_K.p. \tag{6}$$

$$\sum_{i=1}^{L} sr_i.q > \sum_{i=1}^{K} br_i.q. \tag{7}$$

If and only if f_i exists and (8) and (9) hold, g_i will be generated, and we call sr_F and br_E down critical requests.

$$sr_{F+1}.p < br_E.p < sr_F.p. \tag{8}$$

$$\sum_{i=1}^{E} br_i.q > \sum_{i=1}^{F} sr_i.q. \tag{9}$$

With all the trading requests submitted to AA by SAs and BAs, AA will use the SoD auction rule illustrated in figure 3.

In step (2) and (3), we remove the the selling requests with too high asking prices and rank all the left requests, including demand and supply, in the descending order of price. During the next *do* loop(step (4) to (8)), the first up-crossing point and down-crossing point, along with the corresponding critical requests S_L and br_E, is found, and then the leading selling request S_i whose asking price is under the bidding price of br_E is figured out. To satisfy more buying requests, we replace the critical selling request S_L with the new selling request S_i by shifting the position of S_i to that of S_L. It is motivated by the fact that there are more supply quantities than demand quantities and the buying requests have a greater impact on the total trade quantities.

The *do* loop ends under two conditions. One is that there are no more up-crossing points, which means the entire price-quantity broken line of sellers is below that of buyers. The other is that there is only an up-crossing point but no down-crossing points, which indicates that the price-quantity broken line of buyers is above that of sellers before the up-crossing point while the price-quantity broken line of buyers is below that of sellers after the up-crossing point. The first condition is processed from step (9) to (19). In the first if-else statement(step (10) to (16)), the selling request S_{SN}, and buying request br_{BN}, with the smallest quantity margin between each other, are figured out. After that, buying requests br_i with indices $i < BN$ and sellers with indices $j < SN$ participate in trade(step (17) to (19)). Note that S_{SN} and br_{BN} are sacrificed in order to avoid the partial trade which couldn't meet the QoS needs of trading quantity. Though step (18) and step (19) can also lead to partial trade, this cut of a buyer's demand quantity or a seller's supply quantity is usually so small that it can be neglected without degrading the QoS of auction market. In step (18), if $S_j.q < \sum_{j=1}^{SN-1} S_j.q - \sum_{i=1}^{BN-1} br_i.q/(SN-1)$, seller j sells nothing and the "burden", $\sum_{j=1}^{SN-1} S_j.q - \sum_{i=1}^{BN-1} br_i.q/(SN-1) - S_j.q$, will be split equally by the rest of $SN-2$ sellers. The procedure keeps running until each remaining

(1) collect all the selling requests into S and all the buying requests into B
(2) delete from S the selling requests whose asking prices are higher than the highest bidding price among B
(3) arrange the demand quantities and supply quantities in a descending order of price
(4) **do** until there is no up-crossing point or no down-crossing point
(5) figure out the first up-crossing point and down-crossing point and the corresponding critical requests S_L and br_E
(6) $N = \min\{i | S_i.p \leq br_E.p, L < i \leq |S|\}$
(7) delete all the selling requests $S_j (L \leq j < N)$ from S
(8) **enddo**
(9) **if** there is no up-crossing point
(10) **if** $\sum_{i=1}^{m} br_i.q \leq \sum_{j=1}^{|S|} (S_j.q)$
(11) figure out SN, where $\sum_{i=1}^{SN-1} S_i.q < \sum_{i=1}^{m} br_i.q \leq \sum_{i=1}^{SN} S_i.q$
(12) $BN := m$
(13) **endif**
(14) **else** figure out BN, where $\sum_{i=1}^{BN-1} br_i.q < \sum_{i=1}^{|S|} S_i.q \leq \sum_{i=1}^{BN} br_i.q$
(15) $SN := |S|$
(16) **endelse**
(17) **if** $\sum_{j=1}^{SN-1} S_j.q \geq \sum_{i=1}^{BN-1} br_i.q$
(18) $br_i (i < BN)$ buy all its quantity $br_i.q$ at price $br_i.p$
$S_j (j < SN)$ sell a quantity $S_j.q - (\sum_{j=1}^{SN-1} S_j.q - \sum_{i=1}^{BN-1} br_i.q)/(SN-1)$ at price $S_j.p$
(19) **else** $S_j (j < SN)$ sell all its quantity $S_j.q$ at price $S_j.p$
$br_i (i < BN - 1)$ buy a quantity $br_i.q - (\sum_{i=1}^{BN-1} br_i.q - \sum_{j=1}^{SN-1} S_j.q)/(BN-1)$ at price $br_i.p$
(20) **else** find the up-crossing point and the corresponding br_K and S_L
(21) **if** $\sum_{j=1}^{L-1} S_j.q \geq \sum_{i=1}^{K-1} br_i.q$
(22) $br_i (i < K)$ buy all its quantity $br_i.q$ at price $br_i.p$
$S_j (j < L)$ sell a quantity $S_j.q - (\sum_{j=1}^{L-1} S_j.q - \sum_{i=1}^{K-1} br_i.q)/(SN-1)$ at price $S_j.p$
(23) **else** $S_j (j < L)$ sell all its quantity $S_j.q$ at price $S_j.p$
$br_i (i < K-1)$ buy a quantity $br_i.q - (\sum_{i=1}^{K-1} br_i.q - \sum_{j=1}^{L-1} S_j.q)/(BN-1)$ at price $br_i.p$

Fig. 3. The SoD auction rule in case of supply over demand

seller trades a positive quantity. Step (19) involves the same situation of buyers. The second condition is handled in the last four steps(step (20) to (23)). Only K buyers and L sellers before the up-crossing point can trade and the processing method is similar to the first one.

Case II: Demand over Supply (DoS). In this case, there are more demand quantities than supply quantities available in the auction market. Then (10) holds.

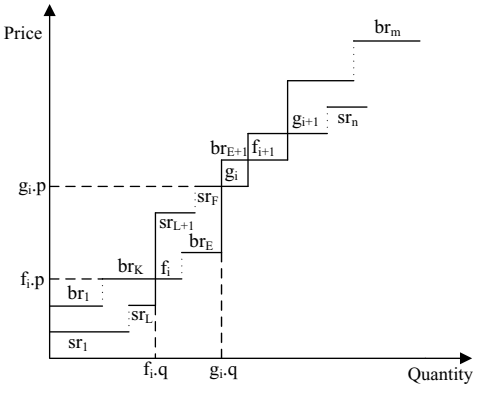


Fig. 4. Demand over Supply

$$\sum_{j=1}^{n} sr_j.q + T < \sum_{i=1}^{m} br_i.q \ (T > 0). \tag{10}$$

Without loss of generality, we assume

$$br_1.p \leq br_2.p... \leq br_m.p. \tag{11}$$

$$sr_1.p \leq sr_2.p... \leq sr_n.p. \tag{12}$$

This time, a DoS auction rule will be adopted by GDAM. The DoS rule is similar to the SoD rule in **Case I** except that the buying requests will be shifted to satisfy more selling requests since the selling requests have a greater impact on the total trade quantities and should be satisfied first. According to the DoS auction rule, GDAM arranges the demand quantities and supply quantities in the ascending order of price(refer to figure 4). We can see from figure 4, the price-quantity broken line of the buyers crosses that of the sellers at some crossing points. The up-crossing point f_i occurs when (13) and (14) hold, and we call br_K and sr_L up critical requests.

$$sr_L.p < br_K.p < sr_{L+1}.p. \tag{13}$$

$$\sum_{i=1}^{K} br_i.q > \sum_{i=1}^{L} sr_i.q. \tag{14}$$

If and only if f_i exsits and (15) and (16) hold, g_i will be generated, and we call sr_F and br_F down critical requests.

$$br_E.p < sr_F.p < br_{E+1}.p. \tag{15}$$

$$\sum_{i=1}^{F} sr_i.q > \sum_{i=1}^{E} br_i.q. \tag{16}$$

The DoS auction rule is shown in figure 5.

(1) collect all the selling requests into S and all the buying requests into B
(2) delete from S the selling requests whose asking prices are higher than the highest bidding price among B
(3) arrange the demand quantities and supply quantities in the ascending order of price
(4) **do** until there is no up-crossing point or no down-crossing point
(5) figure out the first up-crossing point and down-crossing point and the corresponding critical requests B_K and sr_F
(6) $N = \min\{i | B_i.p \geq sr_F.p, K < i \leq |B|\}$
(7) delete all the buying requests $B_j (K \leq j < N)$ from B
(8) **enddo**
(9) **if** there is no up-crossing point
(10) **if** $\sum_{i=1}^{|B|} B_i.q \leq \sum_{j=1}^{n} (sr_j.q)$
(11) figure out SN, where $\sum_{i=1}^{SN-1} sr_i.q < \sum_{i=1}^{|B|} B_i.q \leq \sum_{i=1}^{SN} sr_i.q$
(12) $BN := |B|$
(13) **endif**
(14) **else** figure out BN, where $\sum_{i=1}^{BN-1} B_i.q < \sum_{i=1}^{n} sr_i.q \leq \sum_{i=1}^{BN} B_i.q$
(15) $SN := n$
(16) **endelse**
(17) **if** $\sum_{j=1}^{SN-1} sr_j.q \geq \sum_{i=1}^{BN-1} B_i.q$
(18) $B_i (i < BN)$ buy all its quantity $B_i.q$ at price $B_i.p$
$sr_j (j < SN)$ sell a quantity $sr_j.q - (\sum_{j=1}^{SN-1} sr_j.q - \sum_{i=1}^{BN-1} B_i.q)/(SN-1)$ at price $sr_j.p$
(19) **else** $sr_j (j < SN)$ sell all its quantity $sr_j.q$ at price $sr_j.p$
$B_i (i < BN - 1)$ buy a quantity $B_i.q - (\sum_{i=1}^{BN-1} B_i.q - \sum_{j=1}^{SN-1} sr_j.q)/(BN-1)$ at price $B_i.p$
(20) **else** find the up-crossing point and the corresponding up critical requests B_K and sr_L
(21) **if** $\sum_{j=1}^{L-1} sr_j.q \geq \sum_{i=1}^{K-1} B_i.q$
(22) $B_i (i < K)$ buy all its quantity $B_i.q$ at price $B_i.p$
$S_j (j < L)$ sell a quantity $sr_j.q - (\sum_{j=1}^{L-1} sr_j.q - \sum_{i=1}^{K-1} B_i.q)/(SN-1)$ at price $sr_j.p$
(23) **else** $sr_j (j < L)$ sell all its quantity $sr_j.q$ at price $sr_j.p$
$B_i (i < K-1)$ buy a quantity $B_i.q - (\sum_{i=1}^{K-1} B_i.q - \sum_{j=1}^{L-1} sr_j.q)/(BN-1)$ at price $B_i.p$

Fig. 5. The auction rule in case of demand over supply

Case III: Supply equals Demand. In this case, the overall supply quantities are equivalent or nearly equivalent to the overall demand quantities. And (17) holds.

$$|\sum_{j=1}^{n} sr_j.q - \sum_{i=1}^{m} br_i.q| < T \ (T > 0). \tag{17}$$

Our mechanism is very simple. The SoD rule in **Case I** and the DoS rule in **Case II** are used in turn. The one that enables a larger trade amount will be finally adopted.

4 Features of GDAM

Double auctions have some features, which can also be evidenced in our mechanism, such as strategy-proof, weakly budget-balanced, etc.

> ***Theorem 1:*** *Under the assumption that the QoS requirements(prices and quantities) of the buyers and sellers are private informations, GDAM is strategy-proof with respect to both QoS parameters, weakly budget-balanced and individually rational.*

Proof: According to GDAM, the price-quantity broken line of the traded sellers is always below that of the buyers. Assume M and N are the collection of these buyers and sellers respectively, we can get $\sum_{i=1}^{|M|} br_i.p * br_i.q - \sum_{j=1}^{|N|} sr_j.p * sr_j.q \geq 0$ which indicates that the mechanism can always get nonnegative payment. Hence our mechanism is weakly budget-balanced. In fact, the nonnegative surplus is supposed to be earned by the market maker for managing the auction market.

As stated in section 3, a bidding price covers the profit expected by the buyer and the asking price contains the profit expected by the seller. Each agent gets expected profit if it succeeds in trading, or zero if it fails. Hence our mechanism is individually rational.

Suppose in the scenario of **Case I**, a seller sr_L *over-reports* his asking price in order to get more profit, while others remain unchanged(refer to figure 6). In figure 6 if seller L reports his asking price honestly, he will get the surplus, $ss_L = (sr_L.p - sc_L)sr_L.q$ according to (2). If he over-reports his price by an amount of d, his surplus becomes $(sr_L.p + d - sc_L)sr_L.q$ with an extra surplus of $d \cdot sr_L.q$. If he continues over-reporting the price by another amount e with $sr_L.p + e > br_E.p$, he will lose the opportunity of making successful trade based on SoD rule and gets a surplus of zero. Though over-reporting the price may produce extra surplus, it is hard for a SA to implement this over-reporting

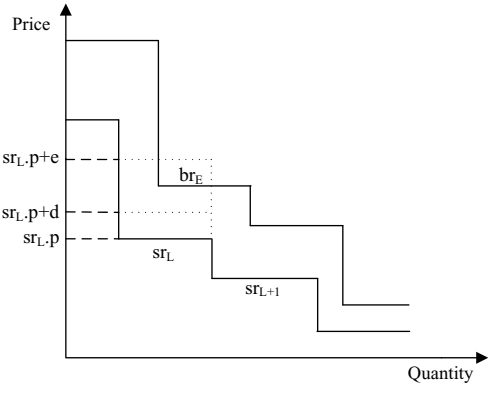


Fig. 6. Seller L over-reports his price while others remain unchanged

strategy successfully under our mechanism. It is easy to explain. Since the QoS requirements of trading agents are transparent between each other, a SA never knows which agents will trade finally, let alone the BA who is "just above" him. Therefore a seller can not decide how much to over-report his price, and the arbitrary over-reporting may actually reduce his surplus as discussed previously. The same proving procedure can be applied to the scenario of **Case II** where a buyer wants to under-report his price. Since the trading agents have no idea of which auction rule (SoD or DoS) will be adopted by our mechanism and misreporting their prices may take a big risk of losing profit, they will not cheat on their prices. Furthermore, an agent has no incentive to misreport his trading quantity since an agent's trading price will not be affected by the quantity under GDAM. So we can see that GDAM is strategy-proof with respect to both price and quantity.

Theorem 2: *GDAM enables larger trade amount than the traditional mechanism, resulting in a better utilizing of grid resources.*

Proof: Assume that K buyers and L sellers will finally trade under the traditional auction mechanism. If we arrange these L sellers in the descending order of price, as illustrated in figure 7, we can see that these L sellers will also succeed in trading according to SoD rule. Moreover, SoD results in a larger trading amount. It works as follows. Seek for the buyer br_E, which satisfies $br_E.p < sr_1.p \leq br_{E-1}.p$. If br_E does not exist, we move forward to find the buyer br_F, which satisfy $br_F.p < sr_2.p \leq br_{F-1}.p$. Continue the process until the proper buyer is found. Without loss of generality, assume br_E is found. Let $g = \sum_{i=1}^{E-1} br_i.q - \sum_{j=1}^{L} sr_j.q$. If there is a seller $sr_H (H > L)$ that satisfies $sr_H.q < g$ and will not cause an up-crossing point (will not let (6) and (7) hold), sr_H, which is abandoned under traditional mechanism, can also trade according to SoD rule. The analysis for DoS rule can be conducted in the same way by arranging the K buyers in the ascending order of price.

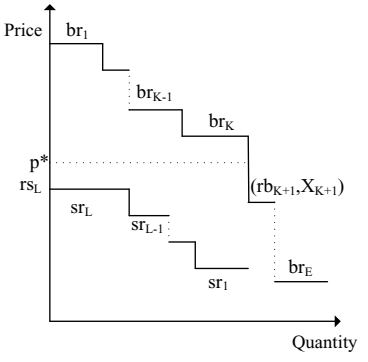


Fig. 7. The L successful sellers are rearranged

5 Experiments and Analysis

5.1 Parameter Definition

The parameters that will be used are given as follows:

- **Number of Agents:** We consider two cases. When the number of SA, n equals the number of BA, m, we run simulation at $m = n$=10, 40, 70, 100, 200, 500, 800, 1000 respectively. When $n \neq m$, the simulation is performed at (m, n)=(20,180), (40,160), (60,140), (80,120), (100,100), (120,80), (140,60), (160,40), (180,20) respectively.
- **Resource Quantity:** We assume that both the resource quantities demanded by buyers and that supplied by sellers are uniformly distributed between 10 and 100 (the unit depends on the kind of resource).
- **Resource Price:** We assume the sellers' asking prices are uniformly distributed between 50 and 100 (Grid $) and the buyers' bidding prices are subject to uniform distribution $U(25, 75)$, $U(50, 100)$ and $U(75, 125)$ respectively.
- **Expected Profit:** As for seller j, his expected profit, sep_j, from per unit of resource equals the result of subtracting the resource cost from his asking price. As for buyer i, if we subtract his bidding price from the value that he can create from consuming per unit of the resource, we get his expected profit, bep_i.
- **Aggregate Surplus** as: Assume M and N represent the collection of buyers and sellers that can trade respectively. Then the aggregate surplus of the auction market is $\sum_{i=1}^{|M|} bep_i \cdot br_i.q + \sum_{j=1}^{|N|} sep_j \cdot sr_j.q$.
- **Market Surplus** ms: The market surplus is the profit that the market can obtain. Assume M and N represent the collection of buyers and sellers that can trade respectively. Let vp be the total value paid by all the buyers, va be the total value got by all the sellers. We have $va = \sum_{i=1}^{|M|} br_i.p \cdot br_i.q$, $vp = \sum_{j=1}^{|N|} sr_j.p \cdot sr_j.q$ and $ms = va - vp$.

5.2 Performance Metric

The following performance metrics have been used for evaluation.

- **Economic Efficiency Loss** eel: In the economic field, a market's efficiency is usually measured by comparing the aggregate profit made by the participants to the maximum profit that theoretically could be made. So under GDAM, the efficiency of the double auction market equals $\frac{as}{as+ms}$. We treat the profit obtained by the market maker, ms as an economic loss and the economic efficiency loss can be expressed as $eel = \frac{ms}{as+ms}$, then the market's economic efficiency is equal to $1 - eel$.
- **Trade Amount** ta : The trade amount, ta is the total quantity of resources that can be successfully traded under our mechanism. A lager trade amount means a greater utilization of grid resources enabled by the market.

Table 1. The auction results under different mechanisms

		$br.p = U(25, 75)$			$br.p = U(75, 125)$		
	$n = m$	ta	SN	BN	ta	SN	BN
TAM	10	64.1	2	2	325.1	6	8
SoD	10	114.5	3	5	452.3	8	9
DoS	10	121.9	3	5	466.1	9	9
TAM	40	479.2	9	11	1578.5	32	36
SoD	40	783.1	15	16	2076.6	37	39
DoS	40	800.1	15	17	2090.4	38	39
TAM	70	892.9	16	17	2817.3	55	47
SoD	70	1515	24	25	3693.2	68	60
DoS	70	1533.7	24	26	3728	69	63
TAM	100	1302.6	21	22	4052.9	68	63
SoD	100	2263.9	42	41	5320.6	98	92
DoS	100	2284.8	42	44	5335.7	99	90
TAM	200	2680.7	40	43	8178.1	144	158
SoD	200	4827.5	79	83	10760.3	189	199
DoS	200	4849.4	75	84	10760.2	189	199
TAM	500	6802.4	124	128	20561.3	382	377
SoD	500	12700.7	232	234	27151	496	494
DoS	500	12725.6	229	235	27187.1	499	495
TAM	800	10926.2	184	200	32915.7	576	590
SoD	800	20692.8	352	375	43549.5	767	798
DoS	800	20718.7	345	376	43565.5	774	796
TAM	1000	13673.1	267	261	41169.5	751	758
SoD	1000	26024.7	493	479	54498.8	996	993
DoS	1000	26151.4	487	484	54514.9	998	990

– **User Satisfaction Percentage:** Under GDAM either a seller or a buyer is the user of the trading service provided by the auction market. The sellers' satisfaction percentage, ssp of the trading service is measured by comparing the number of sellers who trade to the number of all the sellers who participate the auction. The same meaning holds true for the buyers satisfaction percentage, bsp.

5.3 Experiment Procedures

The first experiment was performed to compare GDAM with the traditional auction mechanism(TAM), actually the one proposed in [13]. The number of the buyers, m is equivalent to the number of the sellers, n and the bidding prices' distribution varies from $U(25, 75)$ to $U(75, 125)$ while the asking prices' distribution is fixed at $U(50, 100)$. The experiment result is shown in table 1, where SN and BN represent the number of the sellers and buyers that can trade respectively. All the experiment results in this paper are the average of 10,000 same experiments.

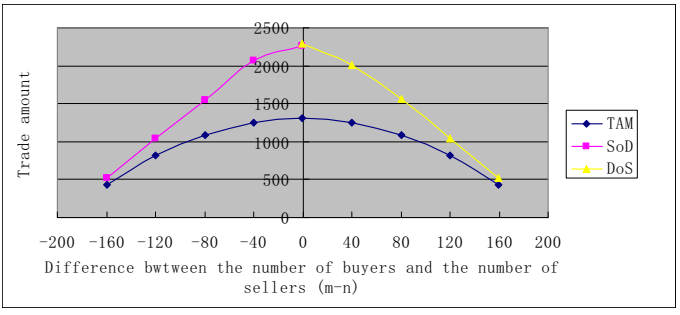


Fig. 8. The auction results with the varying difference between m and n

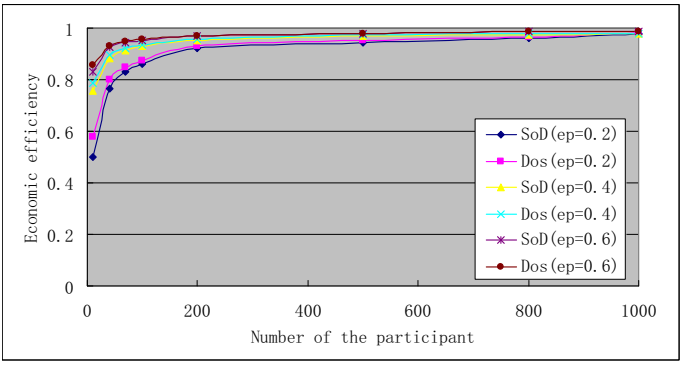


Fig. 9. The economic efficiency of SoD and DoS with $m = n$ and $br.p = U(50, 100)$

We can see when the distribution of bidding price changes from $U(25, 75)$ to $U(75, 125)$, under each of the three mechanisms both the trade amount and the number of successful agents increase. This is easy to understand for a BA with a higher bidding price has more probability to trade. When $br.p = U(25, 75)$ with the same m, the trade amount and the number of successful agents produced by TAM are much less compared to SoD. The trade amount made by TAM only takes up an average of 56.1% of the amount produced by SoD (with a minimum of 53% and a maximum of 61%). This complies with **Theorem 2**. And SoD satisfies much more users (with the average of *ssp* and *bsp* being 40% and 44% respectively) compared to TAM (with the average of *ssp* and *bsp* being 23% and 24% respectively). When $br.p = U(75, 125)$ we can draw a similar conclusion and the user satisfaction percentage is nearly 100%. We can also find that there is merely a slight difference between the results under SoD and DoS in terms of trade amount and user satisfaction percentage, so we can select one out of the two auction rules in random for **Case III**.

In the second experiment we use the scenario where $n > m$ to simulate **Case I** and $m > n$ to simulate **Case II**. The adopted (m,n) pairs are (20,180), (40,160), (60,140), (80,120),(100,100), (120,80), (140,60), (160,40), (180,20) and we assume

the bidding price has a distribution of $U(25, 75)$. When $m - n < 0$ the SoD rule is used to decide which agents will trade ; when $m - n > 0$ the DoS rule is adopted. The experiment result is illustrated in figure 8. We can see that with the value of $m - n$ increasing from -160 to 0 the trade amount increases under either mechanism. The trade amount of SoD is always above the trade amount of TAM and this distance becomes larger when m approaches n little by little. When $m = n$ both SoD and DoS produce the maximum trade amount (2258.9 and 2289 respectively). When $m > n$ DoS is adopted and as the value of $m - n$ increases from 0 to 160, the trade amount decreases from 2289 to 514.7. Even the demand quantity is not equivalent to the supply quantity, GDAM still have an advantage over TAM in improving the utilization of grid resources and when this equivalence becomes smaller the advantage becomes more obvious.

We have also performed an experiment to study the economic efficiency of GDAM. Assume M and N represent the collection of buyers and sellers that can trade respectively. Then the market's economic efficiency loss, $eel = \frac{ms}{as+ms} = \frac{\sum_{i=1}^{|M|} br_i.p \cdot br_i.q - \sum_{j=1}^{|N|} sr_j.p \cdot sr_j.q}{\sum_{i=1}^{|M|} bep_i \cdot br_i.q + br_i.p \cdot br_i.q + \sum_{j=1}^{|N|} sep_j \cdot br_j.q - sr_j.p \cdot sr_j.q}$. For simplicity, we assume the seller's expected profit equals the buyers', and we assume the expected profit, ep is 0.2, 0.4 and 0.6 respectively. The value of economic efficiency of GDMA (1-eel) is shown in figure 9. We can see from figure 9 that as the number of the participants becomes larger the economic efficiency of our mechanism approaches 100% nearer. So GDAM is suitable for the environment where there are a huge number of participants, which can well meet the requirement of the grid. Note that as ep increases from 0.2 to 0.6 the corresponding economic efficiency of both SoD and DoS also increases. This is because successful agents with a larger ep get more surpluses, making the economic efficiency loss relatively smaller.

6 Conclusion

Traditional double auction mechanisms aim at maximizing the total market value instead of enabling more participants to benefit from the market, which is to some extent a kind of system-centric allocation mechanism. In this paper, a novel greedy double auction mechanism(GDAM) is proposed, under which both SAs and BAs consume the trading service that can meet their own QoS requirements of the expected price and amount. What is more important, to satisfy as more participants as possible, GDAM makes use of SoD and DoS rules in different cases. The advantages of GDAM have been proved, such as strategy-proof and individually rational. Simulation results also confirm that GDAM outperforms the traditional one on both the total trade amount and the user satisfaction percentage. And as more agents join in the auction market, the economic efficiency of GDAM will also increase making it is very suitable for the grid environment.

Acknowledgments. This work is supported by the Development Grant for Computer Application Technology Subject jointly Sponsored by Beijing Municipal Commission of Education and Beijing Jiaotong University(XK100040519).

References

1. Buyya, R., Abramson, D., Giddy, J.: A case for economy grid architecture for service oriented grid computing. In: IPDPS 2001: Proceedings of the 15th International Parallel & Distributed Processing Symposium, Washington, DC, USA, p. 83. IEEE Computer Society, Los Alamitos (2001)
2. Buyya, R., Abramson, D., Venugopal, S.: The grid economy. Proceedings of the IEEE, 698–714 (2005)
3. Sutherland, I.E.: A futures market in computer time. Communications of the ACM 11(6), 449–451 (1968)
4. Buyya, R.: Economic-based distributed resource management and scheduling for grid computing. PhD thesis, Monash University, Melbourne, Australia (2002)
5. Regev, O., Nisan, N.: The popcorn market - an online markets for computational resources. In: ICE 1998: Proceedings of the First International Conference on Information and Computation Economies, pp. 148–157. ACM, New York (1998)
6. Waldspurger, C.A., Hogg, T., Huberman, B.A., Kephart, J.O., Stornetta, S.W.: Spawn: A distributed computational economy. IEEE Transactions on Software Engineering 18(2), 103–117 (1992)
7. Bubendorfer, K.: Fine grained resource reservation in open grid economies. In: E-SCIENCE 2006: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing, p. 81. IEEE Computer Society, Los Alamitos (2006)
8. Lalis, S., Karipidis, A.: Jaws: An open market-based framework for distributed computing over the internet. In: GRID 2000: Proceedings of the First IEEE/ACM International Workshop on Grid Computing, London, UK, pp. 36–46. Springer, Heidelberg (2000)
9. Lai, K., Rasmusson, L., Adar, E., Sorkin, S., Zhang, L., Huberman, B.A.: Tycoon: an implemention of a distributed market-based resource allocation system. Technical report, HP Labs (2004)
10. Satterthwaite, M.A., Williams, S.R.: The rate of convergence to efficiency in the buyer's bid double auction as the market becomes large. The Review of Economic Studies 54(6), 477–498 (1989)
11. Williams, S.R.: Existence and convergence of equilibria in the buyer's bid double auction. The Review of Economic Studies 58(2), 351–374 (1991)
12. Yokoo, M., Sakurai, Y., Matsubara, S.: Robust double auction protocol against false-name bids. In: ICDCS 2001: Proceedings of the The 21st International Conference on Distributed Computing Systems, Washington, DC, USA, pp. 137–145. IEEE Computer Society, Los Alamitos (2001)
13. Huang, P., Scheller-wolf, A., Sycara, K.: Design of a multiunit double auction e-market. Computational Intelligence 18(4), 596–617 (2002)
14. Kant, U., Grosu, D.: Double auction protocols for resource allocation in grids. In: ITCC 2005: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC 2005), Washington, DC, USA, vol. I, pp. 366–371. IEEE Computer Society, Los Alamitos (2005)
15. Liu, Y., He, H.C.: Multi-unit combinatorial auction based grid resource coallocation approach. In: SKG 2007: Proceedings of the Third International Conference on Semantics, Knowledge and Grid, Washington, DC, USA, pp. 290–293. IEEE Computer Society, Los Alamitos (2007)

16. Gonen, R., Lehmann, D.: Optimal solutions for multi-unit combinatorial auctions: branch and bound heuristics. In: EC 2000: Proceedings of the 2nd ACM Conference on Electronic Commerce, pp. 13–20. ACM, New York (2000)
17. Chien, C.H., Chang, P.H.M., Soo, V.W.: Market-oriented multiple resource scheduling in grid computing environments. In: AINA 2005: Proceedings of the 19th International Conference on Advanced Information Networking and Applications, Washington, DC, USA, pp. 867–872. IEEE Computer Society, Los Alamitos (2005)
18. Wieczorek, M., Podlipnig, S., Prodan, R., Fahringer, T.: Applying double auctions for scheduling of workflows on the grid. In: SC 2008: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, Piscataway, NJ, USA, pp. 1–11. IEEE Press, Los Alamitos (2008)

Risk Aware Overbooking for Commercial Grids

Georg Birkenheuer, André Brinkmann, and Holger Karl

Paderborn Center for Parallel Computing (PC^2),
Universität Paderborn, Germany
{birke,brinkman,holger.karl}@uni-paderborn.de

Abstract. The commercial exploitation of the emerging Grid and Cloud markets needs SLAs to sell computing run times. Job traces show that users have a limited ability to estimate the resource needs of their applications. This offers the possibility to apply overbooking to negotiation, but overbooking increases the risk of SLA violations. This work presents an overbooking approach with an integrated risk assessment model. Simulations for this model, which are based on real-world job traces, show that overbooking offers significant opportunities for Grid and Cloud providers.

1 Introduction

Grid, Cloud, and HPC providers need intelligent strategies to optimally utilize their existing resources, while not violating quality of services (QoS) guarantees negotiated with the customers and described through service level agreements (SLAs). For the acceptance test of committing to an SLA, a provider uses runtime estimations as well as a deadline from the customer. Job traces show that the user's ability to estimate runtimes is limited[1]. This leads to a statistical measurable overestimation of runtimes as well as to underutilized resources, as jobs are tending to end earlier than negotiated.

To increase the resource utilization and therefore the profit of a provider, we propose to combine overbooking and backfilling techniques for parallel resources in the acceptance test. This instrument should increase system utilization, while not affecting already planned jobs. To successfully use overbooking strategies, we have to be able to calculate the risk of violating SLAs. Our approach uses a history of the distribution of job execution time estimations and their corresponding real runtimes. The probability of success (PoS) for overbooking can then be calculated based on the likelihood that the job finishes within the given runtime.

Scheduling Model. We propose a commercial scenario, where a job execution is negotiated between a Grid customer and a provider. For operation, the grid provider uses a planning based scheduling system. This means that the jobs are not scheduled in a queue, but added to a plan of jobs, where each job has, if accepted, an assigned start time, a number of assigned resources, and a maximum duration. The scenario has four characteristics:

E. Frachtenberg and U. Schwiegelshohn (Eds.): JSSPP 2010, LNCS 6253, pp. 51–76, 2010.

1. the underlying scheduling strategy is FCFS with conservative backfilling
2. the user pay for their submitted jobs proportionally to the computation time they estimate
3. the customers have to receive their jobs' results within a given deadline.
4. the monetary penalty inflicted on the system's owner for missing a job's deadline is equal to the price users pay to for a successful execution

Under these assumptions, the our approach evaluates whether schedulers can exploit automated runtime predictions (along with the fact users typically give inaccurate runtime estimates) in order to *overbook* the gaps within the schedule in a manner that increases the overall profit of the provider.

Technically, the plan of a scheduler has two dimensions, where the width is the number of nodes in the cluster and the time corresponds to the height. When a job request enters the system, the scheduler puts the job in this plan as a rectangle between its release time and deadline. If the job is accepted, it is placed in the schedule and no other jobs can be assigned to this area (see Figure 1). Therein, the latest start is the point in time, where all previous jobs ended consuming their whole estimated runtime. Here the job can start in every case as long the underling resources did not crash. The earliest possible start time is the time where the job can start if all previous jobs needed null runtime. This practically means the start of the latest job starting before or the jobs release time. The job will thus start somewhere in between the earliest and

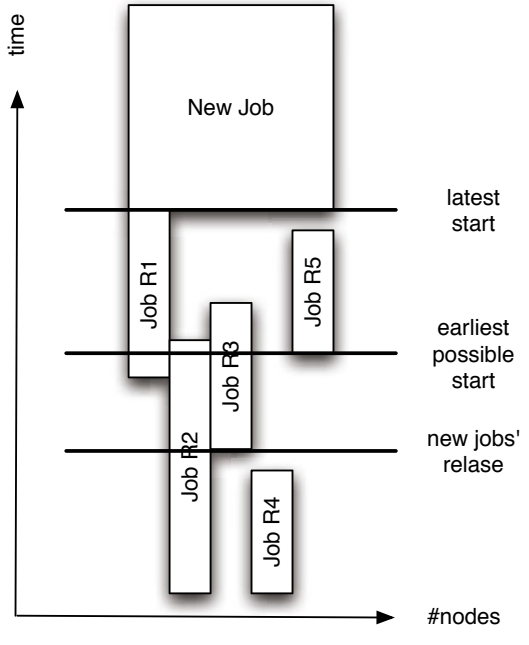


Fig. 1. Exemplary job schedule

latest possible start. If the scheduling algorithm cannot place the job according to the resource and deadline constraints, the job has to be rejected.

This paper is organized as follows: Section 2 presents the related work on overbooking, scheduling, resource stability assessment, and planning. Section 3 introduces our methods and instruments to measure the inaccuracy of the users' runtime estimates. Section 4 describes how we calculate the probability of success of overbooking a schedule. Section 5 evaluates the proposed methods and algorithms and presents simulation results based on real job traces and the paper finishes with a discussion about the achieved goals.

2 Related Work

This section presents the technical basics and related work in the area of scheduling and overbooking. Firstly, it discusses theoretical approaches for planning, followed by scheduling approaches. Then, the paper introduces related work on machine failures and risk assessment in Grid and Cluster systems. At last, we discuss related work on overbooking and its impact on planning and scheduling.

Planning Theory. Planning strategies are also known as *Strip Packing* problem [2]. The aim is to pack jobs in a way that the height of the strip is minimal while the jobs must not overlap themselves. Different strategies have been developed, which should pack as optimal as possible, where optimal packing itself is NP-hard.

Strip Packing distinguishes between offline and online algorithms. Offline algorithms are unusable in our scheduling approach, as jobs are not known in advance. Approaches usable in our online scheduling environment are bottom-left algorithms, which try to put a new job as far to the bottom left of the strip as possible [3]. Level algorithms split the strip horizontally in levels of different sizes [4]. In these levels, appropriate sized jobs can be placed. Shelf algorithms divide the strip vertically in smaller shelves, which could be used for priority based scheduling [5]. Hybrid algorithms are combinations of the above-mentioned algorithms [2]. The disadvantage of the presented scheduling approaches is that jobs in Grid or Cloud environments are connected with an SLA that contains a strict deadline. Therefore, the approach of strip and shelf algorithms of packing jobs earlier or later is impossible. A usable approach here is the simple bottom-left algorithm, where the bottom is given by the earliest start time of a job and a natural ceiling is given by its deadline.

Scheduling Approaches. Many scheduling strategies for cluster systems are still based on first-come first-serve (FCFS). FCFS guarantees fairness, but leads to a poor system utilization as it might create gaps in the schedule.

Backfilling, in contrast, is able to increase system utilization and throughput [6]. It has not to schedule a new job at the end of a queue, but is able to fill gaps, if a new job fits in. The additional requirement for the ability to use backfilling is an estimation about the runtime of each job. The runtime estimations are,

in our scenario, part of the SLAs. The *EASY* (Extensible Argonne Scheduling sYstem) backfilling approach can be used to further improve system utilization. Within EASY, putting a job in a gap is acceptable if the first job in the queue is not delayed [6]. However, EASY backfilling has to be used with caution in systems guaranteeing QoS aspects, since jobs in the queue might be delayed.

Therefore, Feitelson and Weil introduced the *conservative* backfilling approach, which only uses free gaps if no previously accepted job is delayed [7]. Simulations show that both backfilling strategies help to increase overall system utilization and reduce the slowdown and waiting time of the scheduling system [8]. The work also shows that the effect of the described backfilling approaches is limited due to inaccurate runtime estimations. Several papers analyzes the effect of bad runtime estimations on scheduling performance.

An interesting effect is that bad estimations can lead to a better performance [9]. Tsafrirs shows an approach to improve scheduling results by adding a fixed factor to the user estimated runtimes [10].

Effort has been taken to develop methods to cope with bad runtime estimations. Several approaches tried to automatically predict the application runtimes based on the history of similar jobs [11,12,13]. Tsafier et al. present a scheduling algorithm similar to the EASY approach (called EASY++) that uses system-generated execution time predictions and shows an improved scheduling performance for jobs' waiting times [14]. The approach shows that automatically runtime prediction can improve backfilling strategies.

The approaches found in literature are not directly applicable to our work. The algorithms target queuing based systems and provide best effort. Their aim is to improve system utilization and to decrease the slowdown of single jobs. Our approach is a planning based scheduling scenario with strict deadlines, given by SLAs. We want to provide an acceptance test, where we have to decide if we can successfully accept an additional job and thus improve a provider's profit by overbooking resources.

Machine Failure and Risk Assessment. Schroeder [15] and Sahoo [16] have shown that machine crashes in cluster systems are typically busted and correlated and Iosup and Nurmi showed that the failures rates of large clusters follows a Weibull distribution best [17,18]. The project AssessGrid [19] created instruments for risk assessment and risk management at all Grid layers. This includes risk awareness and consideration in SLA negotiation [20] and self-organisation of fault- tolerant actions. The results allow Grid providers to assess risk and end-users also to know the likelihood of an SLA violation in order to accurately compare providers SLA offers. The motivation of the research presented in this paper has its origin in work done by AssessGrid.

Overbooking. Overbooking is widely used and analyzed in the context of hotels [21] or airline reservation systems [22,23]. However, overbooking of Grid or Cloud resources differs from those fields of applications. A cluster system can always start jobs if enough resources are free, while a free seat in an airplane cannot be occupied after the aircraft has taken off.

Overbooking for web and Internet service platforms is presented in [24]. It is assumed that different web applications are running concurrently on a limited set of nodes. The difference to our approach is that we assign nodes exclusively. Therefore, it is impossible to share resources between different applications, while it is possible to use execution time length overestimations, which are not applicable for web hosting.

Overbooking for high-performance computing (HPC), cloud, and grid computing has been proposed in [25,26]. However, the references only mention the possibility of overbooking, but do not propose solutions or strategies. In the Grid context, overbooking has been integrated in a three-layered negotiation protocol [27]. The approach includes the restriction that overbooking is only used for multiple reservations for workflow sub-jobs. Chen et al. [28] use time sharing mechanisms to provide high resources utilization for average system and application loads. At high load, they use priority-based queues to ensure responsiveness of the applications. Sulisto et. al [29] try to compensate no shows of jobs with the use of revenue management and overbooking. However they do not deal with the fact that jobs can start later and run shorter than estimated.

Nissimov and Feitelson introduced a probabilistic backfilling approach, where user runtime estimations and a probabilistic assumption about the real end time of the job allow to use a gap smaller than the estimated execution time [30]. In the scope of estimating the PoS of putting a job in a gap, the probabilistic backfilling and our overbooking scenario are similar. The difference is that Nissimovs acceptance test is applied to an already scheduled job and aims to reduce its slowdown, while our approach is used during the acceptance test at arrival time [30].

We have proposed our ideas for overbooking with focus on a single resource [31,32] and are extending the algorithms and investigations for parallel resources in this work.

3 Probability Density Function

The Probability Density Function (PDF) for a job describes the likelihood that a job ends after exactly x % of its estimated runtime. An example for a PDF is given in Figure 2, which shows this probability distribution for all jobs submitted to a compute cluster in 2007.

Building a PDF. We assume that a job has an assigned start time, but it can start anytime after its release time, if the corresponding resources are free earlier. Therefore, all currently running jobs on the assigned resources have an influence on the real start time of a new job. Every such job has its own probability density function that describes its likelihood to end at some point in time. The aim of the following algorithm is to build a joint PDF, which contains information for a set of jobs. This PDF is the basis to calculate the probability that this set of jobs ends before the deadline of the last job.

The challenge is that several jobs j_1 to j_n can end before the start of a new job. The maximum number n of jobs is equal to the amount of resources required

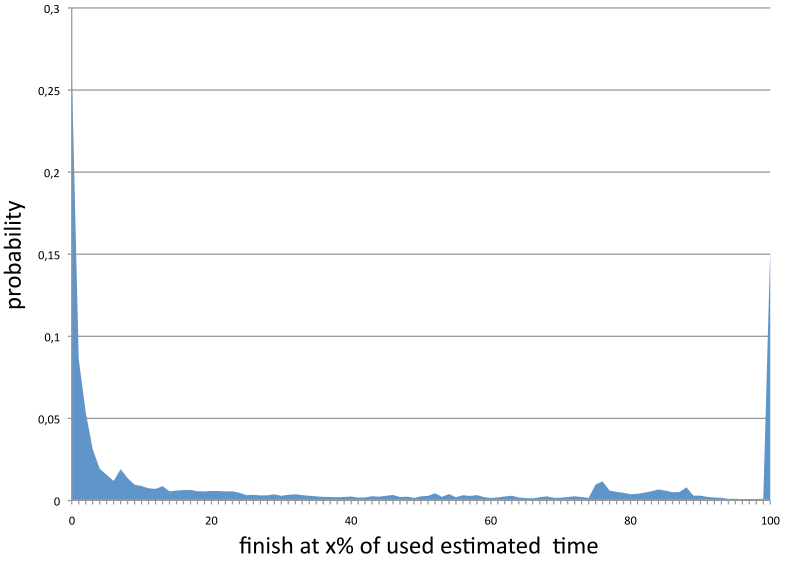


Fig. 2. The PDF derived from all jobs of 2007 in the examined cluster

by the job. The minimum number is zero, when all resources are free at the job's release time.

The latest point in time, where a job will start is the latest planned finish time of any job planned on the used resources before.

However, a job is allowed to start earlier if possible. The earliest possible start time is either the jobs release time or the time where the job can start if all previous need zero runtime. This is the start time of the latest job starting before. The new job might start directly after this job's start, if it ends directly after dispatching, for example due to a missing input.

An example is given in Figure 1 for a job which requires five resources . We have to calculate the PDF of all jobs that are scheduled before and can possibly run in the time between the release and start of the new job.

Deriving PDFs from Job Traces. One way to create PDFs is an analysis of the ratio of real to estimated runtime of historical job traces. Figure 2 shows the Probability Density Function of an exemplary cluster for 2007.

This work assumes that the user's estimation accuracy will not change too much over time. Thus, the past performance of users might be a good estimate for the future. The more job traces are available, the more information the PDF can contain. This work did not only calculate a basic PDF for all jobs, but also different PDFs for different estimated job runtimes.

In Figure 3, we show eight different cumulative distribution functions (CDF) each for an estimated time range, which are integrated over the corresponding PDFs. The figure shows that the estimation quality of the users is best for

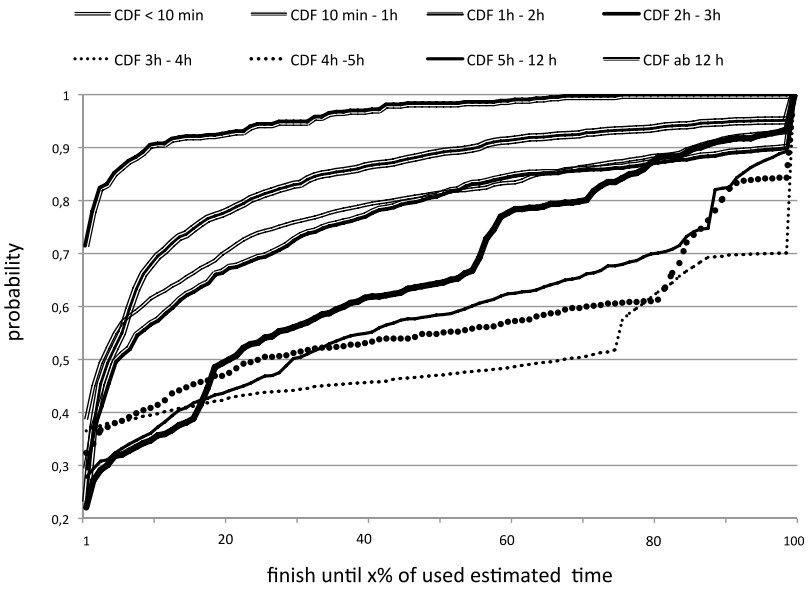


Fig. 3. The CDF for several time slots

jobs from three to four hours. As result, we assume that quality of the users' estimations in our planning based scheduling also depends on the estimated length of the runtime.

Calculating the joint PDF for a Job. We have derived several PDFs for different runtime estimations. Thus, when a new job arrives in the system, the most appropriate PDF, according to the estimated runtime, is chosen for the job itself. Unfortunately, the PDF is not a continuous function, but given from traces. In our framework, we have decided to use discrete steps with one value for each percent step.

For the calculation of the probability that a job ends at time t, it is necessary to calculate the expected joint probability density function for the execution time distribution for the job and its predecessors. In the case that both PDFs are overlapping, the expected joint execution time distribution consists of the convolution of the jobs basic PDF and the calculated PDF of all jobs finishing earlier (see also [32]).

For the simulation, the convolutions are based on discrete values and are stretched or shrank according to the required number. This is given by the number of steps used per time unit and the length of the job. In reality the distributions are continuos functions and the discrete mapping reduces the accuracy. Nevertheless, the convolutions have to be calculated numerically, as no (reasonable) closed formula exists.

4 Risk Assessment for Overbooking

The aim of this section is to define the statistical model used to calculate the probability of success (PoS), which is later assigned to every job. The PoS is calculated based on the statistical runtime overestimations, the estimated runtime for the job, the maximal available runtime inside the gap, as well as the failure rate of the resource. If the gap is smaller than the estimated runtime, there is a chance that the job will still be successful, if it finishes earlier.

The PoS helps the overbooking algorithm to manage and control the underlying scheduling. An acceptance test has to be applied for each new job to decide whether it is beneficial to accept this job even when the underlying resources are overbooked.

Table 1. Job scheduling information

Variable	Content
r	release time
ω	estimated execution time
ddl	deadline
s	start time
f	finish time of the job
n	number of nodes

We assume that the system consists out of N resources, where each resource has the same failure rate λ and repair rate μ. A job j requests n resources and has an earliest release time r, an estimated execution time ω, and a deadline ddl. When the job is placed, the start time s is either its release time or the finish time of the last previous job. The finish time f is important if the scheduling strategy follows conservative backfilling, where the job should not delay following jobs. Therefore, the job will be killed at $f = s_{\text{next}}$.

Calculating the PoS for Overbooking. The probability of successfully completing an overbooked job depends on the probability of resource failures and the probability that the new job finishes in time. To finish in time means that the job has an execution time that fits into a gap between f_{last} and s_{next}. For the calculations, we will define a job j as a tupel $[s, r, \omega, ddl, n]$. The result of the calculation is the probability that a new job is successful in a given gap.

PoS(j_{new}). The probability $\text{PoS}(j_{\text{new}})$ depends on the probability $P_{\text{available}}(s)$ that the requested resources are operational at start time s, the probability $P_{\text{executable}}(j_{\text{new}})$ that the job is able to end within its given maximum execution time, and $P_{\text{success}}(j_{\text{new}})$ which is given by the machine failure rate λ and the job's execution time. Therefore,

$$\text{PoS}(j_{\text{new}}) = P_{\text{available}}(s) \cdot P_{\text{executable}}(j_{\text{new}}) \cdot P_{\text{success}}(j_{\text{new}}).$$

$P_{available}(s)$. The probability that the resource is operational at the start time is

$$P_{\text{available}}(s) = (\frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}})^n = (\frac{\frac{1}{\lambda}}{\frac{1}{\lambda} + \frac{1}{\mu}})^n = (\frac{1}{1 + \frac{\lambda}{\mu}})^n$$

where n is the requested number of resources, MTTF is the mean time to failure $\frac{1}{\lambda}$ and MTTR is the mean time to repair $\frac{1}{\mu}$. This model assumes that the node failures are independent, which is a simplification compared to previous work [17,15]. It has been shown that node failures are bursty and correlated. However, as a job execution is not possible even when one of the planned resources fails, we do not include the amount of other node failures here. In praxis, when the failure rates and behavior of the underling cluster system is known $P_{\text{available}}(s)$ should be analyzed in more detail. However, the failure analysis is not in scope of this work.

$P_{executable}(j_{new})$. The calculation of the probability to successfully execute $P_{\text{executable}}(j_{\text{new}})$ is given by the PDF convolution. The result (between 0 and 1) is the PoS of the job. The higher this value is, the more likely is the success.

If the job j_{new} has no predecessor it is scheduled at its release time and $P_{\text{executable}}(j_{\text{new}})$ is given by its own execution time distribution and the maximal execution time t of the job. $P_{\text{executable}}(j_{\text{new}}) = 1$ if the job has its full estimated execution time ω available and less if the job is overbooked.

If the job j_{new} has one or more direct predecessors the convolution of the execution time distribution has always to be computed with the joint distribution of the previous jobs, which already includes the distributions from all possibly influencing previously planned jobs.

$$P_{\text{executable}}(j_{\text{new}}) = \int_0^t (PDF_{\text{jobs before}} \circ PDF_{\text{new job}})$$

$P_{success}(j_{new})$. $P_{\text{success}}(j_{\text{new}})$ describes the probability that the job's resources survive the execution time. It has been shown that crashes in cluster systems are correlated an bursty [15,16] and the failures rates of large clusters follows a Weibull distribution [17,18]. Following, the definition of $P_{\text{success}}(j_{\text{new}})$ as $1 - e^{-(\frac{x}{\beta})^k}$ would describe the survival rate. Here x is the execution time, $\beta > 0$ describes the spreading of the distribution, and k describes the failure rate over time. A value of $k < 1$ indicates that the failure rate decreases over time, due to hight infant mortality, $k = 1$ means the failure rate is constant, and a value of $k > 1$ indicates that the failure rate increases with time, e.g. due to some aging process.

However, the Weilbull distribution describes an aging processes of the resources over years while the typical jobs are lasting hours to some days. In addition, the failure rate λ has to be adapted over the day and week/weekend as it is shown that it depends on the load of the system [15,16]. As the current workload traces do not contain the corresponding machine failure traces, we concentrate on the job traces and simplify the failure rate. We assume a constant

failure rate λ for the job execution time x. The constant failure rate allows us to model the probability that the job's resources survive the execution time as the constant failure probability λ, the job's execution time x, times the number of requested resources n, and therefore,

$$P_{\text{success}}(j_{\text{new}}) = e^{-\lambda \cdot x \cdot n}.$$

Risk an Opportunity of Overbooking. Mathematically, the opportunity of overbooking would be defined by the PoS of the job and the possible income described as fee of the SLA. On the other hand, there is always a risk accepting an overbooked job. This is defined as the probability occurrence times the impact of this event. The probability of the bad effect is the PoF of accepting the job and the impact is described by the penalty defined in the SLA for a violation.

Accordingly, during SLA negotiation a simple equation can decide whether it is beneficial to accept an SLA with overbooking or not.

- If($PoS \cdot Charge > PoF \cdot Penalty$) accept the SLA,
- else reject the SLA.

This term simply says: Do not accept jobs, where the risk is higher than the opportunity.

Possible Planning Strategies. Generally, the scheduler holds a list of all jobs in the schedule. For each new job j_{new} arriving in the system, the scheduler computes the PoS for the execution of this job in every free space in the schedule where the job might be executed. For the concrete implementation of the scheduling algorithm, several strategies could be applied. A conservative approach could be chosen, where the job is placed in the gap with the highest PoS, a best-fit approach uses the gap providing the highest profit, while still ensuring an acceptable PoF and a first fit approach places the job in the first gap with acceptable PoS.

Implemented First Fit. In this paper we will further investigate an overbooking strategy based on first fit. We check all time-slots starting with the release time of the job where at least the requested amount of resources is available. For each time slot, the algorithm checks, how long the requested resources will be available. If more resources than requested are available, the algorithm chooses the first resources according to their numbers, placing it as left as possible. The algorithm calculates the PoS for placing the new job in this gap based on the chosen resources, the gap length, and the joint PDF. If the PoS is higher than the given threshold, the algorithm places the job in the gap. The approach is thus strongly related to the bottom left first approach in the field of strip packing algorithms [3].

The Overbooking Process. Concluding, the overbooking algorithm follows 5 steps:

1. For every new job
2. Detect the possible places for the job

3. Do for all places beginning with the first
   (a) Calculate the joint PDF for the jobs before
   (b) Combine the PDF of the jobs before and the actual job's PDF.
   (c) With this PDF, the resource stability, and number of resources, build the PoF
   (d) If the PoF is smaller than the threshold, accept the job
4. If no place with suitable PoF has been found, reject the job.

5 Evaluation

This section describes the evaluation of the benefit of our overbooking approach. We have used four job traces from the parallel workloads archive[1] as input, namely SDSC SP2, KTH, BLUE, and CTC. The presented simulations evaluate the outcomes for conservative backfilling and two different overbooking approaches. Firstly, we use a basic statistical model with one PDF built from past user-estimations and secondly, we use an extended statistical model with several PDFs for different time slice lengths.

Simulation Model. Several parameters influence the simulation results. For each test run, the incoming jobs contain the number of required nodes and an estimated and real job length. The job submission times and their release times as well as the up and downtime of the resources have been randomly chosen (see Table 2). Based on this input data, the strategies have been applied and the results are evaluated.

Simulation Resources: Actually, we chose the number of nodes for the simulation according to the size of the cluster system where the traces were from. Thus, the numbers of nodes in the simulation were 128 nodes for the SDSC trace, 100 for KTH, 144 for the BLUE trace, and 430 nodes for CTC. The stability of the underlying resources is not given in the traces. Therefore, the simulation has set the chance to survive a month for each resource to 95%, which correspondents to $\lambda = 0.000068943$ and lasted the MTTR of 12 hours ($\mu = 0.08333$).

Table 2. Job Creation Model

Variable	Description
req. job length e	Chosen from job traces
real job length ω	Chosen from job traces
average time between submission o	1 hour
average delay between job submission and release time r	12 hours
deadline ddl	$r + 5 \cdot e$
req. nodes n	Chosen from job traces

[1] Parallel Workloads Archive:
http://www.cs.huji.ac.il/labs/parallel/workload/

Charge and Penalty. A very important point for the economical adaptability of overbooking is the ratio of the charge of an SLA to its penalty. The overbooking strategy has to be more careful, if ratio between penalty and charge is higher while the opportunity becomes bigger for higher charges. The simulation assumes the charge and penalty are the same and one hour execution time counts as one virtual money unit.

Job Creation Model. The jobs arrival times follow an exponential distribution with given delay to the last job. This delay directly describes the load of the simulation, the faster the jobs are arriving the higher the possible utilization. The chosen simulation parameters enforce that more jobs are submitted than the system could successfully execute. This is done to be able to simulate an environment were overbooking seems to be promising. The release time of the jobs also follows an exponential distribution with a mean of 12 hours which is added to the job submission time. Each simulation ends after the deadline of the last accepted job.

One input parameter of each simulation run is a threshold P_{max} that provides the maximum PoF acceptable by the scheduler for different situations. The overbooking strategy of accepting jobs is based on the PoF given by the convolution of the execution time distribution with the distribution of the previous jobs. A job is placed in the first gap where the calculated PoF is lower than $P_{\max}$.

Use of Different Job Traces. We have removed all jobs that do not contain an estimated runtime as well as a real runtime entry. All jobs except the last $1,000$ trace entries were used for each setting to learn the jobs' runtime behavior. Based on this jobs we created, according to Section 3, a distribution for the simple overbooking approach and several time slice distributions for the time slice overbooking. Thereafter, we have used the last $1,000$ jobs as simulation input.

SDSC. Figures 4 to 7 show the results based on the SDSC SP2 trace. Figures 4 and 5 show the accumulated results of Figures 6, and 7. This means for the jobs Figure 4 contains the successful minus the failed jobs and Figure 5 contains the profit minus the penalty. For this simulation, 0.5 hours have been chosen as basic random value for the delay between the jobs. The SDSC cluster system had 128 nodes. From the $60,000$ jobs of SDSC the first $59,000$ were taken to learn the jobs' runtime behavior. The simulation starts with a maximum acceptable PoF for a job of 0.05 and ends with 1. Like in all following simulation runs, 1000 jobs were submitted to the system.

The backfilling strategy always planed 570 jobs with $2,600$ hours execution time. Both overbooking strategies have at the beginning a sum of 770 jobs and a bit more than $2,600$ hours gain. These 770 jobs are the successful jobs minus the failed jobs.

The number of jobs is for both overbooking approaches at the beginning $P_{\max} = 0.05$ much better than backfilling and then rapidly shrinking. This has two reasons. Firstly, for a higher threshold, jobs with more nodes and longer estimated runtimes are accepted. This circumvented the acceptance of some

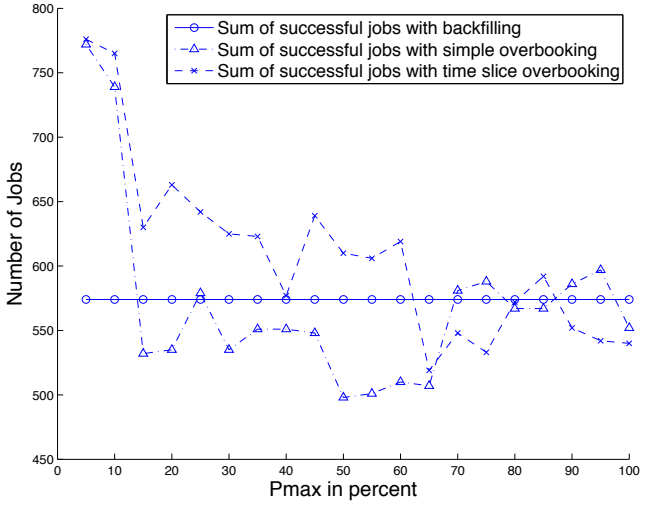


Fig. 4. SDSC: Sum of successful jobs

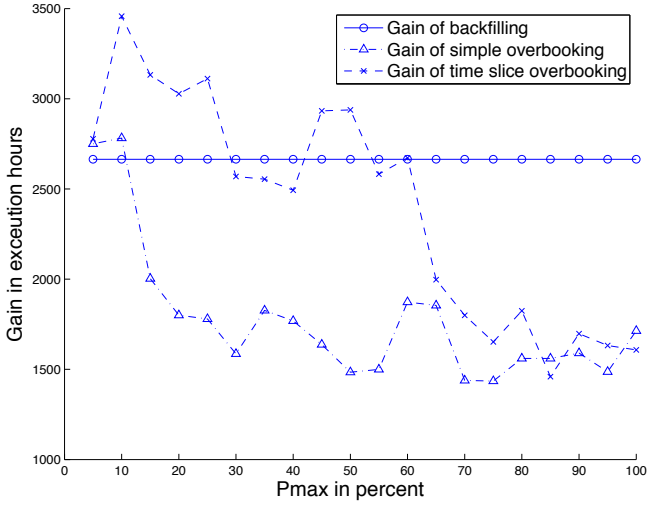


Fig. 5. SDSC: Sum of profit

shorter jobs. Secondly, an increasing amount of jobs failed with the increasing $P_{\max}$.

For the gain, which reflects the successful utilization of the resources, the behavior is a little different. The gain of simple overbooking is at the beginning just a little bit better than the backfilling approach and shrinks for higher $P_{\max}$. This shows that the quality of the underlying statistical analysis is paramount for a successful overbooking approach. The profit for the time slice approach

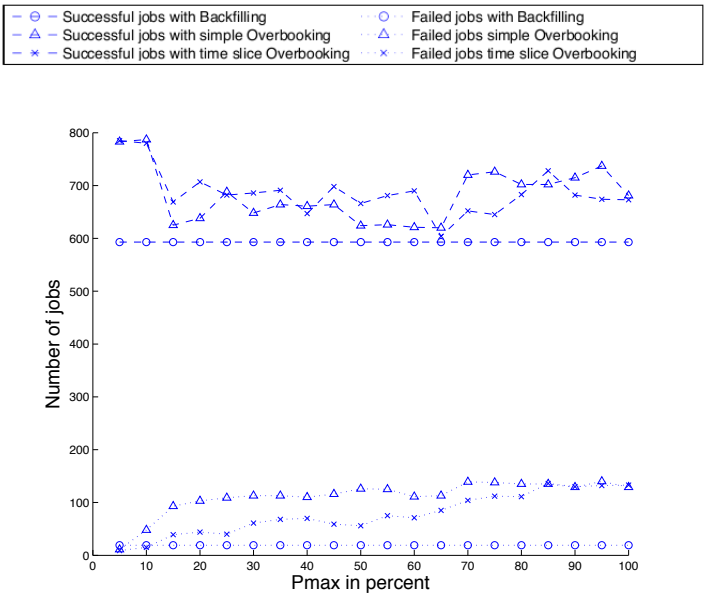


Fig. 6. SDSC: Shown is the number of successful and failed jobs

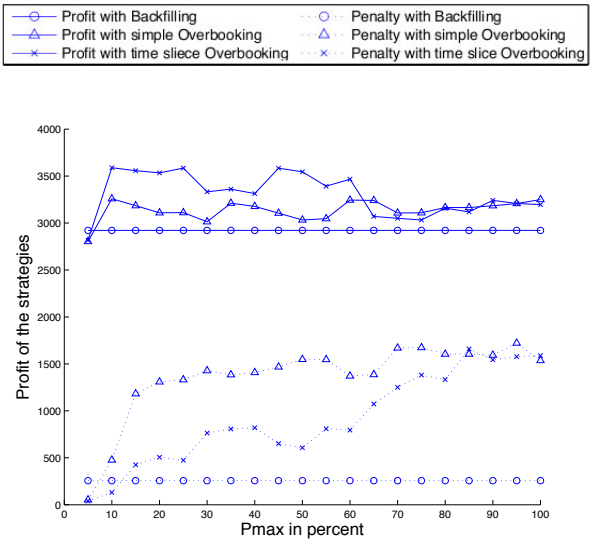


Fig. 7. SDSC: Shown is the profit and penalties of successful and failed jobs

increases from $P_{\max} = 0.05$ to 0.1 where the sum of successful jobs is falling. This is caused by the fact that the simulation has accepted some longer jobs including more nodes, which were not chosen in the run with a lower threshold

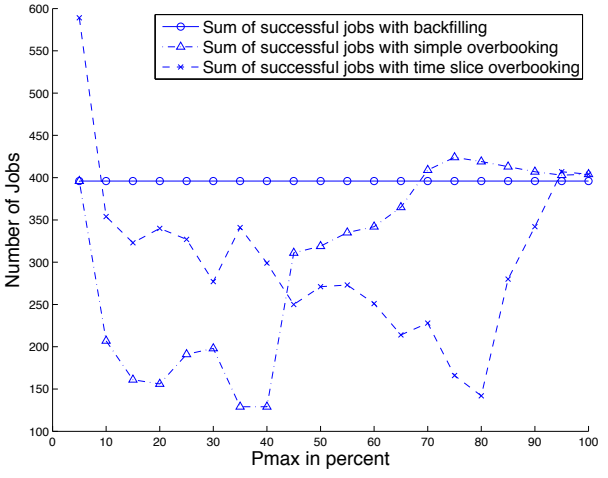


Fig. 8. KTH: Sum of successful jobs

(0.05). Thereafter, the gain of time slice overbooking is falling as more resource consuming jobs are accepted. Some of this jobs are failing hand in hand with the higher accepted risk. This shows that the threshold choice is very important for successfully applying overbooking. The gain of simple overbooking is worse than backfilling from a $P_{\max}$ of 0.1. The time slice overbooking performs better until a $P_{\max}$ of 0.3.

For this simulation, $P_{\max} = 0.1$ should be chosen to maximize profit. With the SDSC traces it is possible to increase the profit by 30 % compared to a conservative backfilling strategy.

All in all, there are many peaks in the figures. This is caused by the fact that with little higher PoF threshold an additional job can be accepted that prohibits the acceptance of some following jobs and vice versa.

KTH. Figures 8, 9, 10, and 11 show the results based on the KTH trace. For this simulation, 0.1 hours have been chosen as basic random value for the delay between the jobs. The KTH cluster had 100 nodes. From the 28,500 jobs of KTH the first 27,500 were taken to learn the jobs' runtime behavior.

The backfilling strategy always planed 400 jobs with 2,100 hours execution time. The simple overbooking strategy has at the beginning also a sum of 400 successful jobs with 1,200 hours gain, while the time slice overbooking has a sum of 600 successful jobs and 2,600 hours gain.

The gain of simple overbooking is nearly always worse than the backfilling strategy. This shows that applying overbooking with a simple statistical analysis can have a severe impact on the providers profit. The sum of successful jobs and gain is falling rapidly under the backfilling level. Interesting is that the number of successful jobs and profit is rapidly shrinking from 0.05 to 0.1 and all in all less jobs are accepted. This means, due to a little higher accepted PoF, jobs with

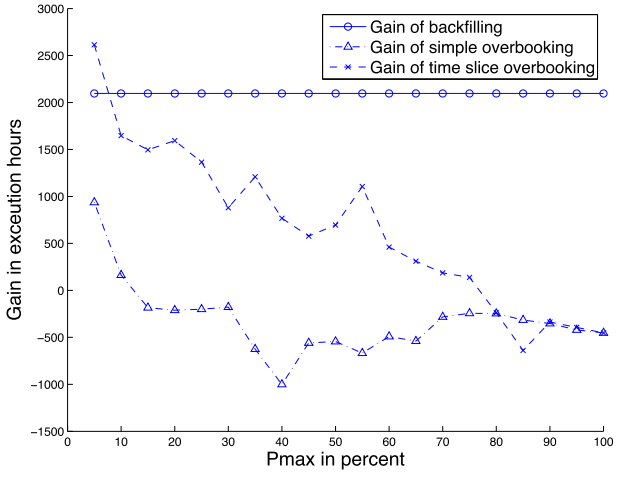


Fig. 9. KTH: Sum of profit

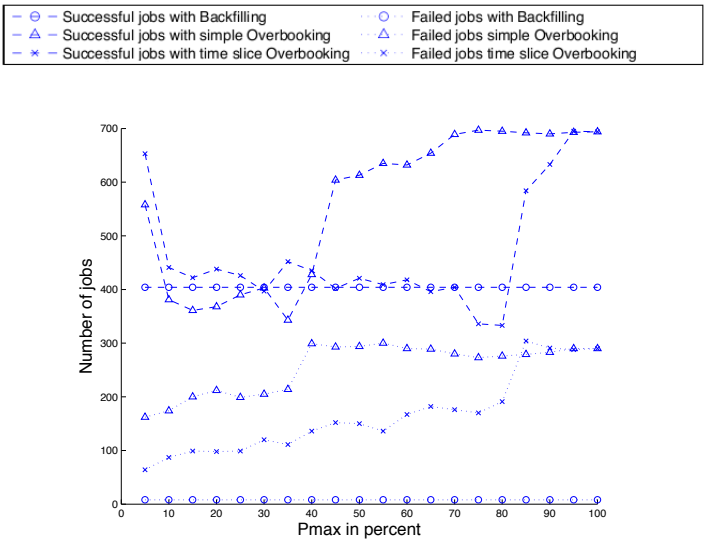


Fig. 10. KTH: Shown is the number of successful and failed jobs

more resource requirements are accepted and fail. With higher $P_{\max}$ the amount of successful jobs is increasing again. This has little effect on the sum of successful jobs as simultaneously the number of failing jobs is also increasing. However, with the use of an improved statistical analysis even with a varying behavior of jobs the overbooking can, carefully adapted, increase the profit. With $P_{\max} = 0.05$ the gain of time slice overbooking is better than the backfilling approach. It

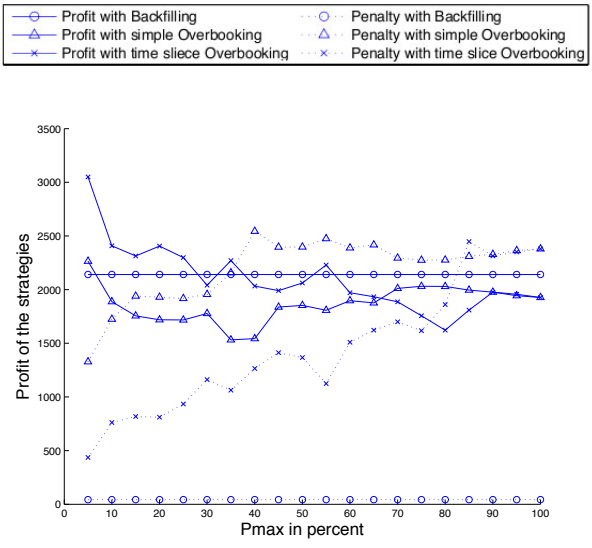


Fig. 11. KTH: Shown is the profit and penalties of successful and failed jobs

is increased by about 23 %. This trace shows that for some user behaviors on clusters an enhanced statistical analysis should be adapted, to further improve the overbooking result. Using statistical analysis based on applications or users basis serve this purpose.

BLUE. Figures 12, 13,14, and 15 show the results based on the BLUE trace. For this simulation, 1 hour has been chosen as basic random value for the delay between the jobs. The BLUE cluster had 144 nodes. From the 243, 000 jobs of BLUE the first 242, 000 were taken to learn the jobs' runtime behavior.

The backfilling strategy always planed 330 successful jobs with an execution time gain of 2, 500 hours. Both overbooking strategies have at the beginning a sum of 690 successful jobs and 2, 500 hours gain. Overbooking strongly depends on $P_{\max}$. For the first simulation runs with low $P_{\max}$ the profit and jobs improves with the increasing $P_{\max}$. From a $P_{\max}$ of 0.2 the sum of successful jobs falls due to less accepted but larger jobs and from a $P_{\max}$ of 0.4 also the gain is falling due to the continuous increasing amount of violated SLAs. Backfilling has more gain than simple overbooking from $P_{\max} = 0.5$ and is better than time slice overbooking from a $P_{\max} = 0.8$. For the BLUE trace and a $P_{\max} = 0.25$, the simple overbooking strategy can increase the gain by 50 % and the time slice overbooking can increase the gain by 55 %.

CTC. Figures 16, 17, 18, and 19 show the results based on the CTC trace. For this simulation, 0.1 hours have been chosen as basic random value for the delay between the jobs. The CTC cluster had 430 nodes. From the 67, 000 jobs of CTC the first 59, 000 were taken to learn the jobs' runtime behavior.

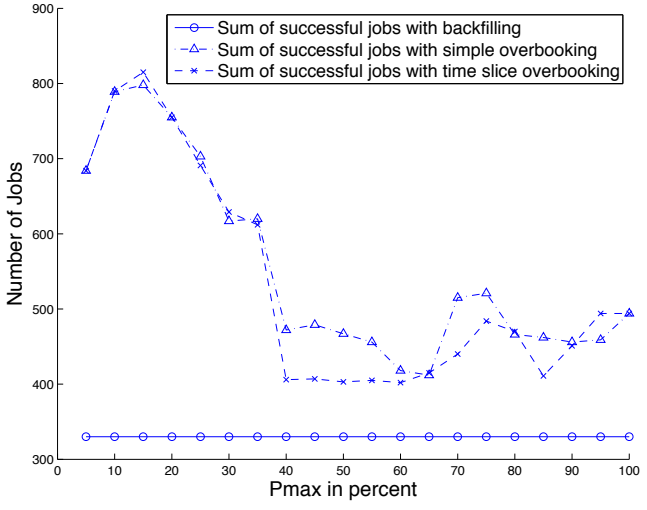


Fig. 12. BLUE: Sum of successful jobs

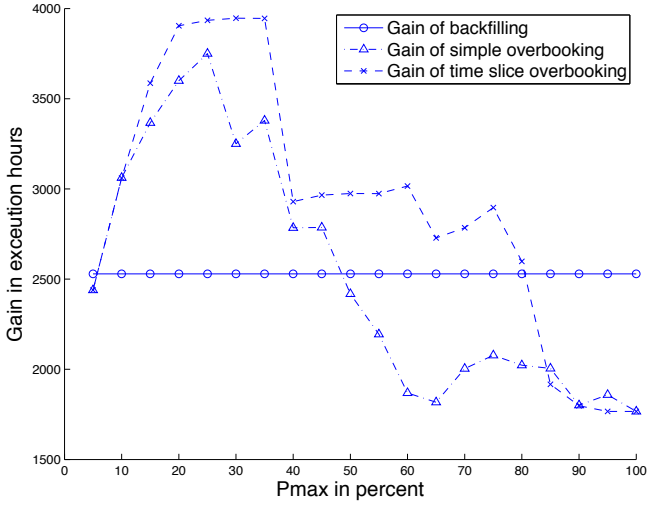


Fig. 13. BLUE: Sum of profit

The backfilling strategy always planed 840 jobs with 7,700 hours execution time. Both overbooking strategies have at the beginning a sum of 930 successful jobs and also 7,700 hours gain. The gain of the simple overbooking approach is maximal for $P_{\max} = 0.1$ and falls under the gain of backfilling from $P_{\max} = 0.15$. The time slice approach produces a maximal gain for $P_{\max} = 0.15$ and is falls

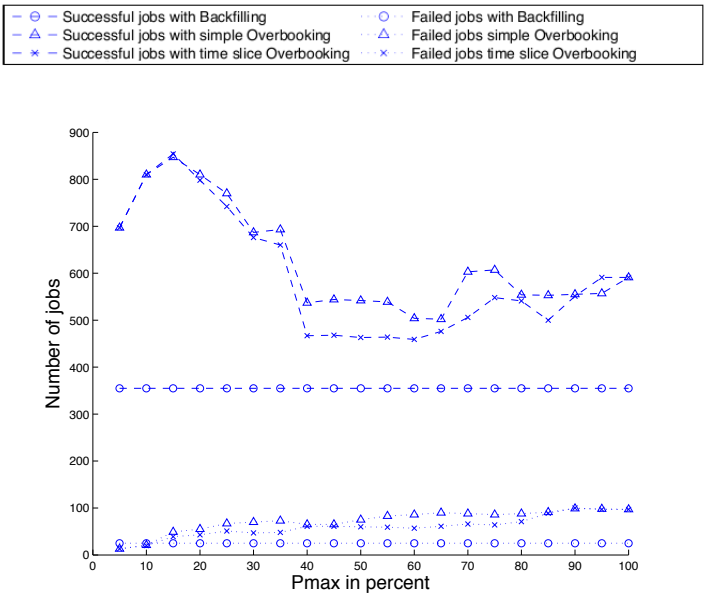


Fig. 14. BLUE: Shown is the number of successful and failed jobs

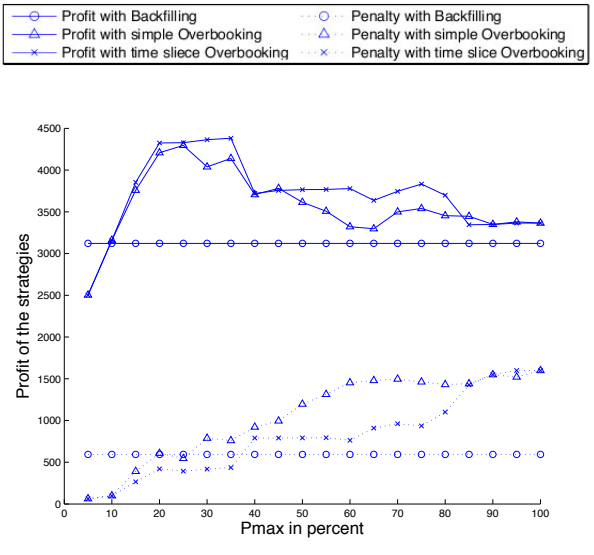


Fig. 15. BLUE: Shown is the profit and penalties of successful and failed jobs

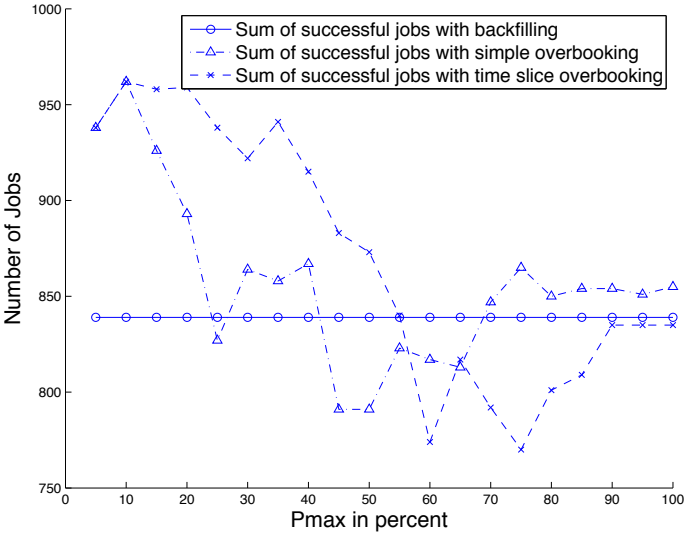


Fig. 16. CTC: Sum of successful jobs

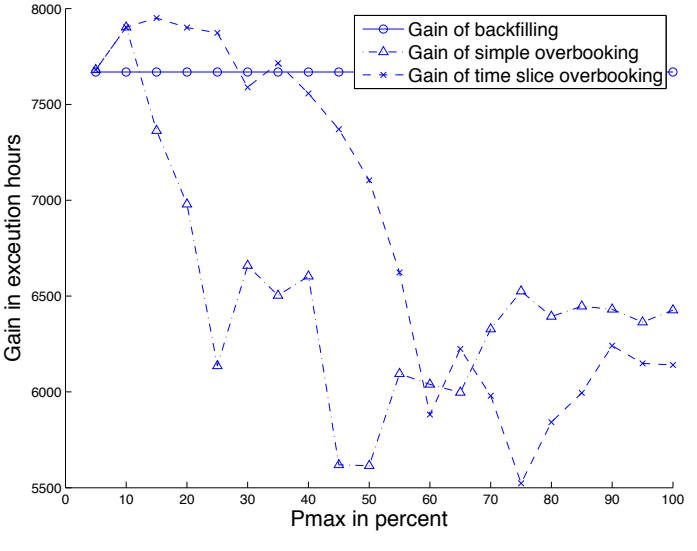


Fig. 17. CTC: Sum of profit

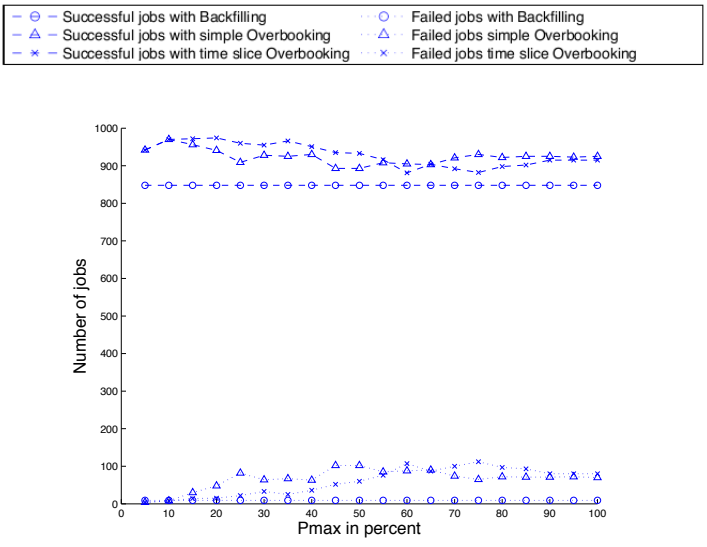


Fig. 18. CTC: Shown is the number of successful and failed jobs

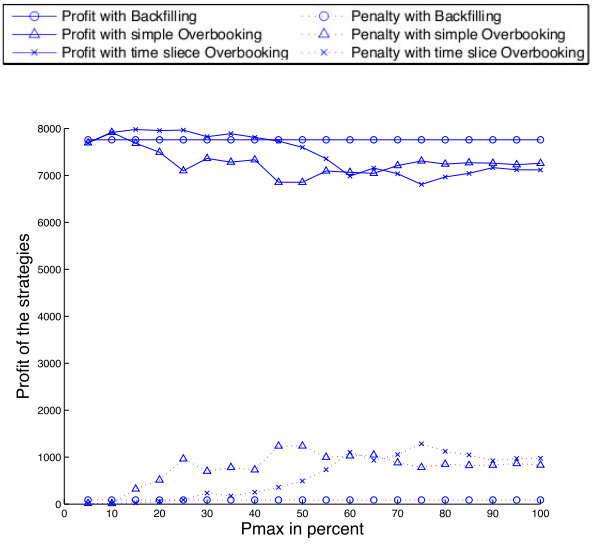


Fig. 19. CTC: Shown is the profit and penalties of successful and failed jobs

under the backfillings' gain from a $P_{\max} = 0.3$. For the CTC trace and a $P_{\max} = 0.1$, the simple and time slice overbooking strategy can increase the gain by 4 %.

CTC with low load. Figures 20 and 21 show the results based on the CTC trace with a low load. For this simulation, 1 hour has been chosen as basic random

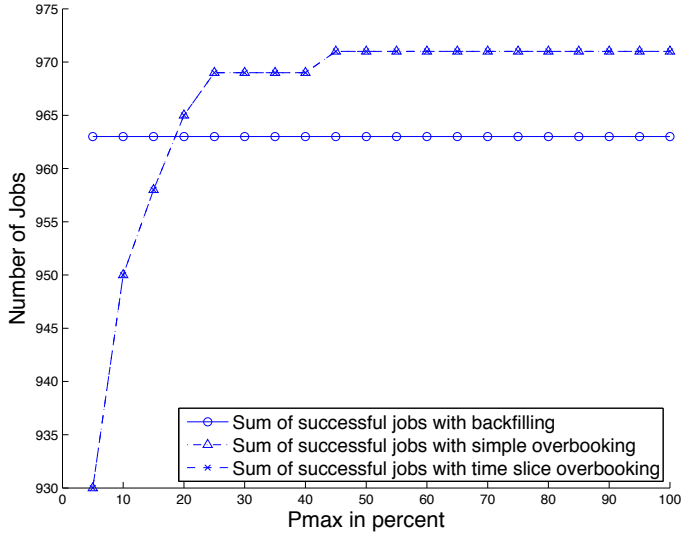


Fig. 20. CTC with low load: Sum of successful jobs

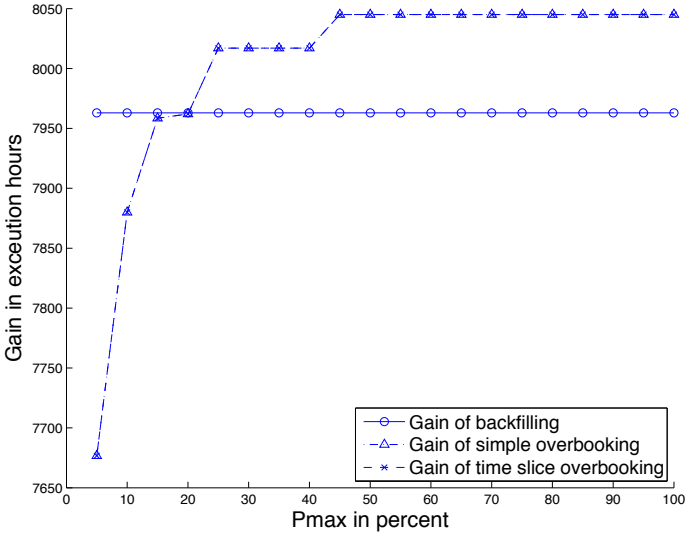


Fig. 21. CTC with low load: Sum of profit

value for the delay between the jobs. The backfilling strategy always planed 978 jobs with 8,000 hours execution time, thus nearly every incoming SLA. We skip the figures for profit/penalty and success/failed jobs here as nothing happens.

The overbooking approaches accept at the beginning less jobs than the backfilling approach. The reason for this behavior is that the risk of machine outages is also calculated in the PoF calculation; this means that for long running jobs including many cores there is a chance that the job might fail due to a machine outage. When the threshold is very low the machine does not accept some of this jobs even if the machine is empty. With a threshold of more than 0.1 the overbooking is similar to backfilling. As no more jobs can be accepted, even accepting high risk, the overbooking profit does not decrease.

6 Discussion

This work aims to increase a providers profit in a commercial scenario, by applying overbooking to resource planning. In the evaluation section we simulated the approach based on job traces from the parallel workload archive.

The simulation underlines that overbooking, carefully applied, provides a good opportunity for a grid provider to further increase its profit. For instance:

- With the SDSC traces and a threshold of $P_{\max} = 0.1$ the profit is increased by 30 % compared to a conservative backfilling strategy.
- With KTH and $P_{\max} = 0.05$ the profit is increased by 23 %.
- With BLUE and $P_{\max} = 0.3$ the profit is increased by 55 %.
- With CTC and a $P_{\max} = 0.15$ the profit is increasing by 4 %

In addition, the evaluation shows that the performance of the time slice overbooking is nearly always better than the simple overbooking. This shows that the quality of the underlying statistical analysis is paramount for a successful overbooking approach.

Where with some traces the profit is increasable by over 50 %, for others only very little additional profit is possible. An improved statistical analysis might still allow to increase the profit, however when the jobs of users in a cluster system (nearly) always fully use then estimated runtime the application of overbooking is not profitable. In addition, the last evaluation shows that the application of overbooking makes sense in cluster systems with high load only.

7 Conclusion

This paper has motivated the idea of using overbooking to increase the ability to accept more SLAs in Grid, Cloud or HPC environments. As overbooking increases the probability of SLA violations, mechanisms for assessing the risk have been shown. The evaluation shows that the additional profit depends on the load of the system, the accuracy of the underlying runtime estimations, and the given real runtime distributions. The additional profit varies depending on the accuracy of the statistical analysis and the load of the system up to over 50 % of additional gain.

For future work it is interesting to determine if there are user and application specific distributions that would allow to increase the quality of the risk estimations for overbooking. Additionally we plan to examine the abilities of using virtualization techniques. This would allow to migrate jobs that took more time as their original gap length allows. If enough other resources are available at the end of a job's gap, the job is moved to these resources, thus an SLA violation might be prevented. Finally, we want to find a heuristic, which can estimate the PoF for a job without the CPU time-consuming convolution of PDF distributions.

References

1. Cirne, W., Berman, F.: A comprehensive model of the supercomputer workload. In: 4th Workshop on Workload Characterization, Citeseer (2001)
2. Hopper, E., Turton, B.: A review of the application of meta-heuristic algorithms to 2D strip packing problems. Artificial Intelligence Review 16(4), 257–300 (2001)
3. Berkey, J., Wang, P.: Two-dimensional finite bin-packing algorithms. Journal of the Operational Research Society, 423–429 (1987)
4. Ntene, N., van Vuuren, J.: A survey and comparison of level heuristics for the 2D oriented strip packing problem. Discrete Optimization (2006)
5. Baker, B., Schwarz, J.: Shelf algorithms for two-dimensional packing problems. SIAM Journal on Computing 12, 508 (1983)
6. Feitelson, D., Jette, M.: Improved utilization and responsiveness with gang scheduling. In: Proceedings of the Job Scheduling Strategies for Parallel Processing: IPPS 1997 Workshop, Geneva, Switzerland, April 5 (1997)
7. Feitelson, D., Weil, A.: Utilization and predictability in scheduling the ibm sp2 with backfilling. In: Proceedings of the 12th International Parallel Processing Symposium (1998)
8. Mu'alem, A., Feitelson, D.: Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp 2 with backfilling. IEEE Transactions on Parallel and Distributed Systems 12(6), 529–543 (2001)
9. Zotkin, D., Keleher, P.: Job-length estimation and performance in backfilling schedulers. In: Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing, HPDC (1999)
10. Tsafrir, D., Feitelson, D.: The dynamics of backfilling: solving the mystery of why increased inaccuracy may help. In: Proceedings of the IEEE International Symposium on Workload Characterization (2006)
11. Gibbons, R.: A historical application profiler for use by parallel schedulers. In: Proceedings of the Job Scheduling Strategies for Parallel Processing (JSSPP): IPPS 1997 Workshop (1997)
12. Smith, W., Foster, I., Taylor, V.: Predicting application run times using historical information. In: Proceedings of the Job Scheduling Strategies for Parallel Processing, JSSPP (1998)
13. Tsafrir, D., Etsion, Y., Feitelson, D.: Modeling user runtime estimates. In: Proceedings of the Job Scheduling Strategies for Parallel Processing, JSSPP (2005)

14. Tsafrir, D., Etsion, Y., Feitelson, D.: Backfilling using system-generated predictions rather than user runtime estimates. IEEE Transactions on Parallel and Distributed Systems (TPDS), 789–803 (2007)
15. Schroeder, B., Gibson, G.: A large-scale study of failures in high-performance computing systems. In: Proc. of the 2006 international Conference on Dependable Systems and Networks (DSN 2006), Citeseer (2006)
16. Sahoo, R., Squillante, M., Sivasubramaniam, A., Zhang, Y.: Failure data analysis of a large-scale heterogeneous server environment. In: 2004 International Conference on Dependable Systems and Networks, pp. 772–781 (2004)
17. Iosup, A., Jan, M., Sonmez, O., Epema, D.: On the dynamic resource availability in grids. In: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing, pp. 26–33. IEEE Computer Society, Los Alamitos (2007)
18. Nurmi, D., Brevik, J., Wolski, R.: Modeling machine availability in enterprise and wide-area distributed computing environments. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 432–441. Springer, Heidelberg (2005)
19. Birkenheuer, G., Djemame, K., Gourlay, I., Hovestadt, M., Kao, O., Padgett, J., Voss, K.: Introducing risk management into the grid. In: Proceedings of the 2nd IEEE International Conference on e-Science and Grid Computing (e-Science 2006), p. 28. IEEE Computer Society, Amsterdam (2006)
20. Djemame, K., Padgett, J., Gourlay, I., Voss, K., Battre, D., Kao, O.: Economically enhanced risk-aware grid sla management. In: Proceedings of eChallenges e-2008 Conference, Stockolm, Sweden (2008)
21. Liberman, V., Yechiali, U.: On the hotel overbooking problem-an inventory system with stochastic cancellations. Management Science 24(11), 1117–1126 (1978)
22. Subramanian, J., Stidham Jr., S., Lautenbacher, C.J.: Airline yield management with overbooking, cancellations, and no-shows. Transportation Science 33(2), 147–167 (1999)
23. Rothstein, M.: Or and the airline overbooking problem. Operations Research 33(2), 237–248 (1985)
24. Urgaonkar, B., Shenoy, P.J., Roscoe, T.: Resource overbooking and application profiling in shared hosting platforms. In: Proceedings of the 5th Symposium on Operating System Design and Implementation, OSDI (2002)
25. Andrieux, A., Berry, D., Garibaldi, J., Jarvis, S., MacLaren, J., Ouelhadj, D., Snelling, D.: Open issues in grid scheduling. UK e-Science Report UKeS-2004-03 (2004)
26. Hovestadt, M., Kao, O., Keller, A., Streit, A.: Scheduling in hpc resource management systems: Queuing vs. planning. In: Proceedings of the Job Scheduling Strategies for Parallel Processing, JSSPP (2003)
27. Siddiqui, M., Villazón, A., Fahringer, T.: Grid allocation and reservation - grid capacity planning with negotiation-based advance reservation for optimized qos. In: Proceedings of the ACM/IEEE SC 2006 Conference on High Performance Networking and Computing, p. 103 (2006)
28. Chen, M., Wu, Y., Yang, G., Liu, X.: Efficiently rationing resources for grid and p2p computing. In: Proceedings of the IFIP International Conference Network and Parallel Computing (NPC), pp. 133–136 (2004)
29. Sulistio, A., Kim, K.H., Buyya, R.: Managing cancellations and no-shows of reservations with overbooking to increase resource revenue. In: Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid), pp. 267–276 (2008)

30. Nissimov, A., Feitelson, D.: Probabilistic backfilling. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2007. LNCS, vol. 4942, pp. 102–115. Springer, Heidelberg (2008)
31. Birkenheuer, G., Hovestadt, M., Kao, O., Voss, K.: Overbooking in planning based scheduling systems. In: Proceedings of the 2008 International Conference on Grid Computing and Applications (GCA), Las Vegas, Nevada, USA (2008)
32. Birkenheuer, G., Brinkmann, A., Karl, H.: The gain of overbooking. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2009. LNCS, vol. 5798, pp. 80–100. Springer, Heidelberg (2009)

The Gain of Resource Delegation in Distributed Computing Environments

Alexander Fölling, Christian Grimme,
Joachim Lepping, and Alexander Papaspyrou

Robotics Research Institute, TU Dortmund University, 44221 Dortmund, Germany
firstname.lastname@udo.edu

Abstract. In this paper, we address job scheduling in Distributed Computing Infrastructures, that is a loosely coupled network of autonomous acting High Performance Computing systems. In contrast to the common approach of mutual workload exchange, we consider the more intuitive operator's viewpoint of load-dependent resource reconfiguration. In case of a site's over-utilization, the scheduling system is able to lease resources from other sites to keep up service quality for its local user community. Contrary, the granting of idle resources can increase utilization in times of low local workload and thus ensure higher efficiency. The evaluation considers real workload data and is done with respect to common service quality indicators. For two simple resource exchange policies and three basic setups we show the possible gain of this approach and analyze the dynamics in workload-adaptive reconfiguration behavior.

1 Introduction

The use of High Performance Computing (HPC) in research, development, and production has become a typical part of day-to-day work since its emergence in the late 1980s; the operation of batch-oriented, large-scale Massively Parallel Processing systems is a commodity service for users in many universities, research centers, and medium-to-large enterprises.

Such systems are typically acquired with respect to the demand of the local user communities. Naturally, this demand is subject to constant change: the usage of HPC systems in research is typically bound to fixed publication dates, and industrial applications depend on the amount of orders or certain—internal and external—projects. Hence, the load of such systems fluctuates over time, a fact that obviously does not comply with the goal of its operator, namely permanent high utilization. In static environments, this leads to two undesirable situations: Either the system is underutilized, which harms the expected return on investment of the HPC system or, in case of over-utilization, the users are forced into unacceptable delays due to a large backlog of work.

From an operator's point of view, the natural way to cope with this tension would be a dynamic reconfiguration of their system in an on-demand fashion: For example, if the user-generated workload grows due to a conference deadline or a project deliverable, the operator would add additional resources to his local

E. Frachtenberg and U. Schwiegelshohn (Eds.): JSSPP 2010, LNCS 6253, pp. 77–92, 2010.

system until the backlog shrinks again. On the other hand, he could offer idle parts of the system to other parts of his organization, such as different departments in an enterprise or cooperating institutes within a network of universities. While such an approach ensures that the system is well-utilized—a fundamental performance benchmark for most operators and their institutions—over time, it also delivers a higher level of service quality to the users due to the adaptiveness to their workload.

The technical foundation for Distributed Computing Infrastructures (DCIs) that are capable of providing such service has been laid during the late 1990s with the emergence of Grid Computing[1]. In this area, a plethora of research has been conducted with respect to sensible workload distribution. Due to the architectural nature of Grid Computing, much effort has been put into mechanisms for the delegation of workload between participating compute centers. However, while being accepted as a basis for very large research projects such as the LHC, Grid Computing is not very wide-spread in the commercial domain and still—due to its stems in academic HPC infrastructures and its strong tailoring to their organizational architectures—comprises a high level of complexity.

Over the last two years, this technical foundation has been largely simplified and commoditized: With the widespread offering of Cloud Computing services and IaaS[1], system administrators can provision additional resources, e.g. compute, storage, and even networking, on-demand without having to make a permanent investment into extending the local facilities.

The availability of such technology in conjunction with the demand for adaptive reconfiguration of DCI environments open new challenges in the management of such systems. With respect to automated capacity planning, the efficient and situation-adaptive scheduling of incoming workload raises interesting questions:

- Is it beneficial for the system owner to invest into an expansion, or would it be sufficient to "lease" a certain amount of resources for a fixed period of time?
- Can the temporary give-away of local resources to befriend departments within a larger company provide both better overall utilization while at the same time ensuring user satisfaction?
- How does the meaning of classic utilization metrics change in such distributed, regularly self-reconfiguring systems?

In this paper, we attempt a first step towards addressing these issues: We assume a simplified DCI scenario with identical resources and investigate the performance of two algorithmic approaches to the leasing and granting of resources between autonomous HPC systems. Herein, we establish mechanisms for situation-based decision making on the distributed management level and evaluate the dynamics of system reconfiguration.

Although scheduling decision making happens mostly on the management level, it comprises to very different aspects in realization: a selection policy to find

[1] Infrastructure as a Service.

adequate partner sites for resource leasing in a distributed scenario as well as the development of decision policies for resource request and delegation respectively. While the former aspect is rather technically addressing balancing behavior on a global level, the latter emphasizes site performance for a local user community.

In order to investigate local behavior, we focus on minimum-sizes scenarios with only two sites and ignore the issue of load balancing on a global level. We evaluate these setups using workload data from three real-world HPC traces, analyze the behavior of resource leases and grants, and find improvements for both user- and provider-related metrics as basis and motivation for further research in resource delegation approaches. Nevertheless, the authors are aware of the fact that these first ideas have to be extended towards scalable heuristics that are capable to deal with mutable partner in a larger DCI.

The remainder of the paper is organized as follows: Section 2 gives an overview of existing approaches to scheduling in DCI environments. This is followed by a formal description of the DCI environment and the resulting scheduling problem in Section 3. Section 4 details our two-layered scheduling architecture while the proposed scheduling policies are then described in Section 5. We present a performance evaluation of our strategies in Section 6 and conclude the paper in Section 7.

2 Background

Automated capacity planning and workload scheduling in DCI systems is a well-covered research topic and stems back to classic parallel machine scheduling problems.

In recent years, a remarkable amount of effort has been put into workload distribution among autonomously acting HPC centers within the broader context of Grid Computing: Scenarios that assume such federated environments often imply centralized scheduling services [2]. For example, Ernemann et al. [3] show advantages of hierarchical scheduling in general by considering the AWRT objective. Further, Kurowski et al. [4] identify multiple objectives for efficient job scheduling in Grids and propose a strategy based on prediction mechanisms and resource reservation. For decentralized environments, only few results that support the delegation of workload have been published. England and Weissman [5] give an estimation of costs and benefits of load sharing relying on synthetic workloads only. Grimme et al. [6] analyze the prospects of collaborative job sharing and compare their results to the non-cooperative scenario of the same machines. Recent works of Fölling et al. [7,8] propose a fuzzy-based, evolutionary optimized exchange policy for a fully decentralized scenario which shows robustness even in changing environments and automatically adapts to the current local load.

With the elasticity of IaaS-supported DCI environments, a new kind of flexibility challenges current scheduling approaches due to the inherent reconfigurability of machines and the resulting changes in scheduling responsibilities. Up to now, this aspect—especially with respect to classic HPC workloads with parallel

jobs—has only occasionally been discussed in research: Since the complexity of operating such systems in the large scale and the feasibility of provisioning and reconfiguring them on-demand hampered the realization for production environments, discussion focused on rather low-level computer hardware. For example, Kota et al. [9] consider the problem of scheduling and mapping of tasks onto reconfigurable logic units for a given application introducing a concept of parameterized modules. Their approach is a typical example of scheduling in the context of reconfigurable hardware that involves varying sizes of available hardware. Subramaniyan et al. [10] transferred similar ideas to the HPC context and analyzed the dynamic scheduling of large-scale HPC applications in parallel reconfigurable computing environments. They assess the performance of several common HPC scheduling heuristics that can be used by an automated job management service to schedule application tasks on parallel reconfigurable systems. However, their approach is limited to a single HPC system and does not involve the interaction of multiple autonomous partners in a DCI environment.

The reconfigurability of a HPC center within a larger DCI environment obviously provides inherent support of multi-site computing on the capacity planning and workload distribution level. In multi-site computing, jobs can be executed beyond site boundaries, effectively running parts of the job at distinct locations. Naturally, additional problems with respect to data availability and network performance arise here. Nevertheless, Ernemann et al. [11] identify improvements for the AWRT objective assuming hierarchical centralized scheduling structures in multi-site computing. Further, Zhang et al. [12] provide an overview of existing multi-site computing approaches and present an adaptive algorithm that incorporates also common local scheduling heuristics. Recently, Iosup et al. [13] proposed a delegated matchmaking method, which temporarily binds resources from remote sites to the local environment.

All approaches assume an additional scheduling layer on top of classic LRMS which coordinates the underlying resources in a hierarchical fashion but their architectures imply that local sites have to (partially) cede their autonomy for the benefit of coordinated DCI scheduling on a higher level.

Further, Weissmann and Grimshaw [14] presented an approach for decentralized DCI systems which introduces all basic policies to exchange jobs between autonomous sites. However, their policies are based on an unrestrictive information model between the sites which allows a local scheduler to query detailed information about the system states of potential delegation targets. This includes also queries on estimated start times at foreign sites for specific jobs. As in real DCI systems such information are usually treated confidential, it requires new heuristics that even yield acceptable scheduling performance when only local information is accessible for decision making. Moreover, their scheduling approaches are only considered theoretically without performance measurement on workload data. Thus, with respect to the work at hand, their results cannot be used for the matter of comparison.

3 Problem Formulation

In HPC systems, job scheduling is an online problem regardless of the assumed machine configurations. Users submit parallel jobs over time while neither their submission time nor the precise processing time are known in advance. We further consider independent[2] jobs that are neither malleable nor moldable. Each job j is characterized by its degree of parallelism m_j, its processor independent processing time p_j and its estimated processing time $\overline{p}_j$, see Feitelson et al. [15]. $\overline{p}_j$ is provided by the user at submit time and originally was intended to recognize erroneous jobs and abort them if they take longer than the user expected. Scheduling heuristics, however, also use $\overline{p}_j$ for making better decisions. The number of required processors m_j is available at the release time r_j of job j.

The DCI environment we consider in our work consists of K loosely coupled HPC sites. Each site k owns m_k identical processors such that every parallel job can be executed on each subset of local processors. Although existing studies for heterogeneous DCI environments show that the processing time of jobs depends on both the application structure and the target architecture, see for example Sabin et al. [16], the list of top-performing HPC installations[3] proves almost homogeneity in terms of processors families and architectures. Therefore, we additionally assume identical processors among all sites.

During its execution phase, each job requires exclusive access to $m_j \leq m_k$ processors. As users submit their jobs locally, the corresponding site has to guarantee that every submitted job can—regardless of the availability of remote systems—be executed. Therefore, jobs that require more than the total number of locally available processors ($m_j > m_k$) are rejected. Further, all jobs run to completion without the possibility of being preempted, since the majority of HPC applications and systems does not support this. As such, the completion time within the schedule S at site k is denoted by $C_j(S_k)$.

In our system model, we further allow multi-site execution, that is each job can be executed on any subset of processors within the whole DCI environment. This is typically possible[4] for embarrassingly parallel jobs that comprise many sequential, independent invocations of the same application. Examples for this application class are parameter sweeps—tools that repeatedly process the same input data, with varying parameter settings—or SPMD[5]-style programs. Iosup et al. [17] have shown that this class is the most widely spread kind of jobs in productive grids and DCI environments. Although distributed filesystem access and network latency may impair the execution speed of such applications in a multi-site execution scenario, Ernemann et al. [11] have shown that the significant improvements in schedule quality often compensate for the inferior performance. Formally, such a multi-site job j is scheduled on $m_{j|k}$ own resources at the submission site $k \in K$ and $m_{j\nmid k}$ foreign resources using altogether $m_j = m_{j|k} + m_{j\nmid k}$ resources as defined before.

[2] With no dependencies among them, that is.
[3] www.top500.org, January 2010.
[4] Provided that data availability (for example, via a shared file system) is guaranteed.
[5] Single Process, Multiple Data.

4 System Model

In our system model, we establish a two-layered architecture at every site, see Figure 1: The Local Resource Management System (LRMS) is responsible for the local allocation of jobs onto resources, while the Distributed Resource Management System (DRMS) layer realizes the the lease-and-grant mechanism and policy. Within the latter layer, resource requests are formulated and negotiated in order to adapt the local system to the current load situation.

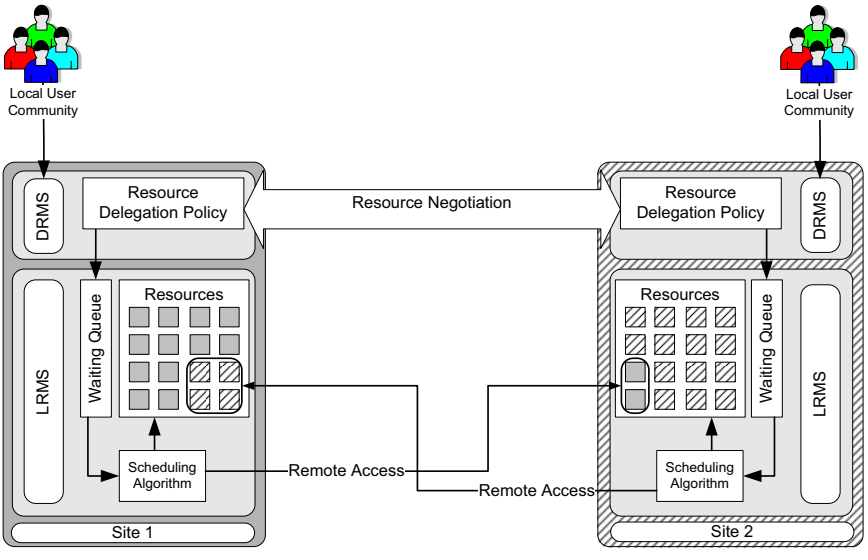


Fig. 1. Resource Brokering within a Computational Grid scenario with independent sites

4.1 LRMS Layer

The Local Resource Management System (LRMS) layer consists of a waiting queue and a scheduling algorithm that assigns jobs to processors in its domain. This *local scheduling domain* comprises all processors that are exclusively controlled by the LRMS. In contrast to classic settings, this domain is subject to changes over time: While foreign resources can be logically integrated into the LRMS and used by the local scheduling algorithm, it is also possible to delegate own resources to *foreign scheduling domains*, putting them under exclusive control of the remote RMS. Further, jobs can be prioritized.

Among the variety of LRMS scheduling algorithms, we chose the Extensible Argonne Scheduling System (EASY) [18] for our analysis as it enjoys widespread application. On invocation, EASY tries to execute the job j at the head of the waiting queue, if—with respect to m_j—enough processors are currently available. Otherwise, it tries to "backfill" a subsequent job in the queue, ensuring that—based on $\overline{p}_j$—job j is not delayed. Note, however, that the overall methodology is not restricted to any kind of local scheduling algorithm.

4.2 DRMS Layer

The DRMS layer is able to extend the local scheduling domain by leasing resources from other sites. In this way, the site is able to gain exclusive control on foreign resources. We assume that submitted jobs have to pass through this layer before they can be handled by the underlying LRMS[6]. For each job, a *resource delegation policy* (RDP) decides whether additional resources should be leased to increase the scheduling performance at the local site or not. If yes, resource requests are send and negotiated with other partners within the DCI system. Each request contains the number of desired resources and a timespan for which the site wants to gain exclusive access to them. Granting sites also apply their RDP in order to determine whether to accept or decline the request. Decision making is based on multiple input features such as the users' job submission behavior, the current resource usage, and the local backlog. Finally, it is not allowed to grant already leased resources to a third party.

5 Resource Delegation Policy

At the DRMS level, the resource delegation policy steers the individual negotiation behavior of each participating site within the DCI. We here introduce two approaches that feature a very simple design, are minimally invasive to the LRMS, and still achieve good scheduling results. Both are triggered by the submission of a single job and can be applied under restrictive information policies, that is without exchange of information between the interacting partners.

The Simple Submission Triggered Resource Delegation (S-STRD) policy tries to prioritize incoming jobs, if a resource lease for this particular submission can be acquired. Figure 2 presents the behavior of the policy in a flow chart (left side) and gives an example for a submitted job (right side). Resource leasing is attempted every time the currently available resources cannot meet the submitted job's processor demand. After the submission of a new job to the DRMS, it is automatically forwarded to and enqueued at the LRMS ①. Furthermore, the DRMS checks whether there are enough idle resources available to directly execute the job ②. In the positive case, the DRMS leaves further handling to the LRMS. Otherwise, the DRMS formulates a resource lease request with the number of requested, but locally unavailable resources and the user's runtime estimate for the job. This request is then posted to the delegation partners in the system ③. If the request is granted, the job is prioritized for direct execution ④. After completion of the job, which is not necessarily equal to the user estimate, leased resources are returned to the granting site.

In the example (cf. Figure 2), we assume a waiting queue with three jobs at Site 1. Further, both sites have scheduled occupations for different timespans. At time t a new job j with a demand of $m_j = 4$ and $\overline{p}_j = 100$ is submitted to Site 1. Since the job cannot be scheduled on local resources directly, Site 1 requests

[6] This poses no restriction in terms of usability, since the DRMS can act as a proxy of the LRMS towards the user.

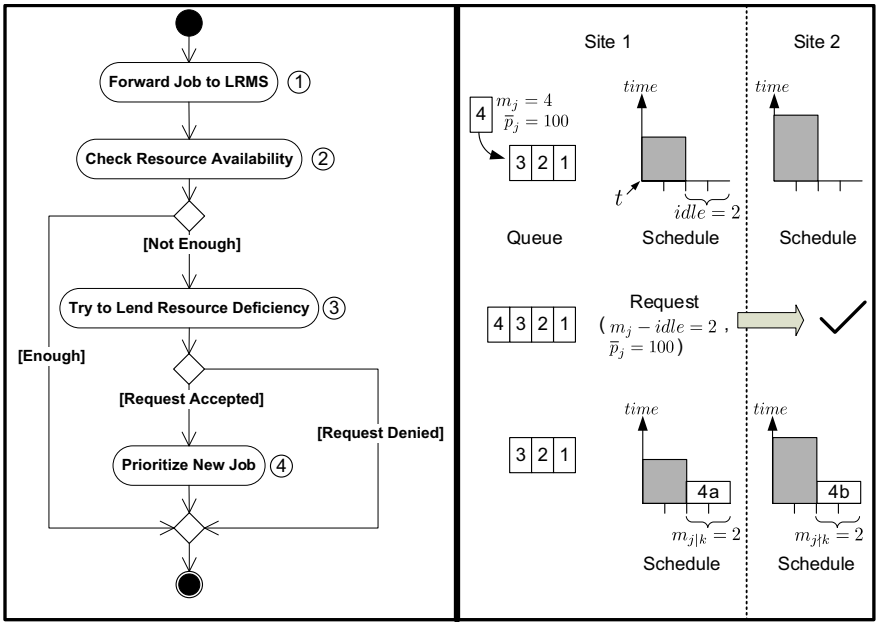


Fig. 2. Activity diagram and example for the Simple Submission Triggered Resource Delegation Policy

two additional resources for a timespan of $\overline{p}_j$ from Site 2, which in turn accepts the request and grants two resources to Site 1. Having a sufficient number of processors available for the immediate execution of j, S-STRD prioritizes the job and, on invocation of the LRMS's scheduling algorithm, it is started immediately on two local and two remote resources.

The extended version of the algorithm (X-STRD) differs in the repetition of steps ②–④ (compare Figure 2): These are applied for each job in the queue (including the new job), starting at the queue's head. Obviously, this approach less penalizes already waiting jobs, since they considered first. Still, this policy demands extensive inter-site communication due to many additional resource requests and makes the extended approach less practical for the use in real scheduling systems. We still evaluate this policy as an extremal case for an excessive use of resource delegation between the sites to assess the achievable performance.

6 Performance Evaluation

We estimate the quality of the proposed mechanisms by means of simulation. In order to quantify the performance, we apply common performance indicators for job scheduling in parallel machine and DCI environments and adapt them to reconfigurable machine environments accordingly. Moreover, we use recorded

(non-synthetic, that is) workload traces as input to our simulations to ensure realistic results. Finally, we discuss the implications from the simulation results for three distinct scenarios.

6.1 Quality Measures

From the quantitative side, we look at three common metrics:

The *Average Weighted Response Time* (AWRT) basically denotes for all users how long they have to wait for their jobs to complete on the average.

$$\mathrm{AWRT}_k = \frac{\sum\limits_{j \in \tau_k} p_j \cdot m_j \cdot (C_j(S) - r_j)}{\sum\limits_{j \in \tau_k} p_j \cdot m_j} \tag{1}$$

It is computed for all jobs $j \in \tau_k$ that have been submitted to site k, see Equation 1. It is widely agreed that a short AWRT is the best way to describe the average performance a provider can offer users for job execution. Following Schwiegelshohn and Yahyapour [19], we weight the response time of each job with its resource consumption $(p_j \cdot m_j)$. This ensures that neither splitting nor combination of jobs can influence the objective function in a beneficial way. Note that we calculate $m_j = m_{j|k} + m_{j\nmid k}$ in order to incorporate the execution of jobs on remote resources.

The *Squashed Area*[7] (SA_k) reflects the overall resource usage of all submitted jobs per participating site k. In a scenario where jobs are partially executed on remote sites, we have to refine the original metric as follows:

$$\mathrm{SA}_k = \sum_{j \in \tau_k} p_j \cdot m_{j|k} + \sum_{l \notin \tau_k} p_l \cdot m_{l|k} \tag{2}$$

SA_k is determined as the sum both local $(j \in \tau_k)$ and foreign $(l \notin \tau_k)$ jobs' resource consumption fractions $(p_j \cdot \ldots$ and $p_l \cdot \ldots)$ that are executed on resources belonging to site k $(m_{j|k}$ and $m_{l|k})$, see Equation 2.

$$\mathrm{SA}_k^{\lambda} = \sum_{j \in \tau_k} p_j \cdot m_{j\nmid k} \tag{3}$$

To further measure the amount of work running on leased processors from within the DCI environment, we define the "leased" Squashed Area SA_k^{λ} as the sum of local $(j \in \tau_k)$ jobs' resource consumption fractions $(p_j \cdot \ldots)$ that are executed on resources not belonging to site k $(m_{j\nmid k})$, see Equation 3.

The *Utilization* (U_k) describes the ratio between overall resource usage available resources after the completion of all and measures how efficiently the processors of site k are used over time.

$$st(S_k) = \min\left\{ \min_{j \in \tau_k}\{C_j(S_k) - p_j\}, \min_{l \notin \tau_k}\{C_l(S_k) - p_l\} \right\}, \text{ and} \tag{4}$$

$$C_{max,k} = \max\left\{ \max_{j \in \tau_k}\{C_j(S_k)\}, \max_{l \notin \tau_k}\{C_l(S_k)\} \right\}. \tag{5}$$

[7] This metric sometime also called *Total Work*.

It refers to the timespan relevant from the schedule's point of view, delimited by the start time of the first job, see Equation 4, to the end time of the last job, see Equation 5, in schedule S_k. Note that both points in time consider local jobs ($j \in \tau_k$) and fractions of delegated jobs ($l \notin \tau_k$).

$$\mathrm{U}_k = \frac{\mathrm{SA}_k}{m_k \cdot (C_{max,k} - st(S_k))} \tag{6}$$

U_k, formally defined in Equation 6, often serves as a quality measure from the site provider's point of view.

6.2 Input Data

The Parallel Workloads Archive[8] provides job submission and execution recordings on real-world HPC system site, each of which containing information on relevant job characteristics like estimated and real processing time, release date, resource demand, and others. We applied pre-filtering steps to the original data in order to remove partially erroneous information: we discard jobs with invalid release dates ($r_j < 0$), processing times ($p_j \leq 0$), resource requests ($m_j \leq 0$) as well as unsatisfiable resource demands on the submitted site ($m_j > m_k$).

Table 1. Workload characteristics of the used input data, including AWRT in seconds, U in %, and C_{max} in seconds for single site execution with EASY

Identifier	#Jobs	m_k	AWRT	U	C_{max}	Setup 1	Setup 2	Setup 3
KTH-11	28479	100	75157.63	68.72	29363626	X	X	
CTC-11	77199	430	52937.96	65.70	29306682	X		X
SDSC05-11	74903	1664	54953.84	60.17	29357277		X	X

We select three traces for our evaluation: The KTH trace which contains records from a 100 processor IBM RS/6000 SP system at the Swedish Royal Institute of Technology in Stockholm, the CTC trace from a 430 processor IBM RS/6000 SP system at the Cornell Theory Center in Ithaca, NY, and a log recorded 2005 at the San Diego Supercomputer Center in La Jolla, CA (SDSC05).

Since the original workloads cover unequal periods, we shorten all original workloads to the largest common lenght, namely eleven months. Additionally, we assume identical timezones and therefore similarize the diurnal rhythm of job submission: In geographically dispersed DCI scenarios, different timezones may induce positive scheduling effects as idle machines can be used by jobs from peak loaded sites in accordance with day-time differences, see Ernemann at al. [20]. Here, we cannot benefit from timezone shifts in our scenario. As such, the results will likely improve in time-shifted environments.

[8] http://www.cs.huji.ac.il/labs/parallel/workload/

Finally, we simulate the workload on their original machine configuration with an non DCI-aware LRMS that uses the EASY algorithm and take the results as reference for local-only scheduling, see Section 4.1. Relevant characteristics of the examined traces and the corresponding results for AWRT, Utilization, and C_{max} are listed in Table 1. During the course of this paper, we will refer to this non-cooperative case for the matter of comparison.

6.3 Performance Results

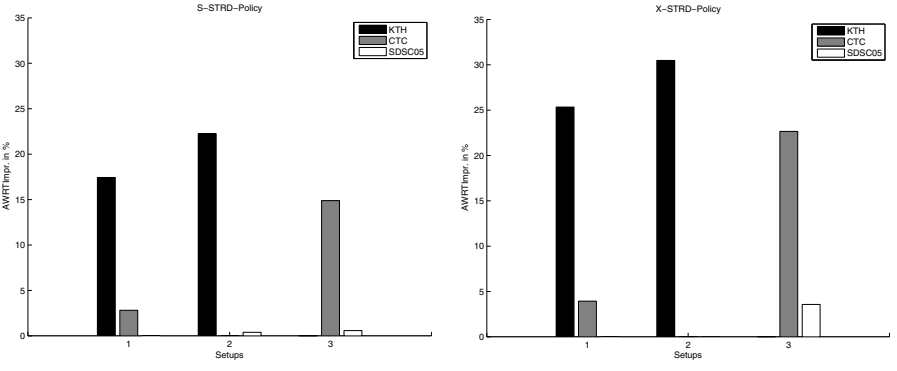
We investigate three scenarios (cf. Table 1): The first scenario comprises the small KTH machine with 100 processors and the mid-sized CTC machine with 430 processors. Further, we evaluate a scenario with the small KTH machine and the large-scale HPC system SDSC05 with 1664 processors and combine further CTC and SDSC05. For all scenarios, we apply the two discussed strategies. The results of all evaluations are shown in Table 2.

Table 2. Evaluation results for both strategies S-STRD and X-STRD for the given three scenarios. Values for AWRT, U, and C_{max} are shown as well as their improvements in %, the absolute amount of mutually exchanged resources as leased Squashed Area SA_k^{λ}, and the average queue length $\bar{Q}$

S-STRD-Broker						
Metrics	Setup 1		Setup 2		Setup 3	
Workload	KTH-11	CTC-11	KTH-11	SDSC05-11	CTC-11	SDSC-11
$AWRT_k$	62055.37	51444.91	58432.61	54738.92	45062.71	54635.66
U_k	65.22	66.46	62.63	60.54	63.37	60.80
$C_{max,k}$	29363626	29332185	29363626	29353826	29328089	29335555
$\Delta AWRT_k$	17.43	2.82	22.25	0.39	14.88	0.58
ΔU_k	-5.09	1.15	-8.86	0.62	-3.54	1.05
SA_k^{λ}	573141728	413432502	630674342	383713739	1768336330	1493028875
$\bar{Q}$	4.08	8.06	2.31	17.07	3.07	13.22
X-STRD-Broker						
Metrics	Setup 1		Setup 2		Setup 3	
Workload	KTH-11	CTC-11	KTH-11	SDSC05-11	CTC-11	SDSC-11
$AWRT_k$	56115.99	50851.43	52253.12	55729.64	40940.23	52990.40
U_k	64.40	66.65	56.88	60.86	62.71	60.98
$C_{max,k}$	29363626	29332185	29363626	29364647	29306682	29339742
$\Delta AWRT_k$	25.34	3.94	30.48	-1.41	22.66	3.57
ΔU_k	-6.28	1.44	-17.22	1.16	-4.55	1.34
SA_k^{λ}	640012760	442455083	682003341	250461072	2105350162	1722880873
$\bar{Q}$	4.48	10.17	1.82	18.62	2.93	14.53

Almost all results show an improvement in AWRT compared to local execution, which indicates that both partners benefit from their cooperation. Figure 3 depicts the improvements obtained in three scenarios for both policies. The S-STRD strategy yields good results, improving the AWRT of the smaller partner for at least 15% in all scenarios. As expected, small partners benefits from

the enormous resource potential provided by large partners. However, simulations show that large partners also profit from cooperation with small partners. Although this improvement is marginal for the SDSC05 site, the increase of utilization, see Table 2, indicates a compact schedule and thus better resource usage.



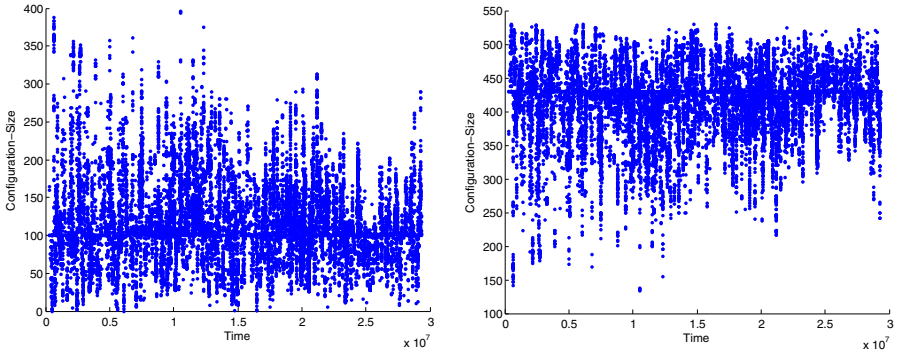
(a) Improvements in AWRT for all setups and S-STRD-Policy.

(b) Improvements in AWRT for all setups and X-STRD-Policy.

Fig. 3. Improvements in AWRT for all setups and both policies

Besides AWRT comparison, we analyze the reconfiguration behavior at both sites. Figure 4 exemplary shows the dynamics of both systems for Setup 1 and the S-STRD policy. Obviously, the local resource configurations are subject to continuous changes while—on the average—they nearly keep their size, see Figure 5. During the simulated workload period, KTH occasionally grants all its resources to the larger site, but also quadruples its original size through leases. In the latter case, the reconfiguration almost switches the original sizes of the setup. This impressively demonstrates the potential of a workload-triggered reconfiguration where the user visible provider domains remain stable: resources adapt to submitted workload but offer an accustomed environment to users.

Finally, we investigate X-STRD and identify larger benefits for all smaller sites. Compared to the application of S-STRD we can also show AWRT improvements for larger sites. However, we observe a deterioration in AWRT for the Setup 2 compared to uncooperative processing, see Figure 3(b). This behavior is due to unbalanced exchange of resources indicated by SA_k^{λ} for the second setup in Table 2: While the small KTH site is able to increase its resource capacity, the larger site's requests are frequently rejected for S-STRD leading to higher utilization and increased AWRT. Thus, we conclude that the extended strategy can yield better results for all participating sites but is less robust against large discrepancies in machine size: In X-STRD, continuous workload submission results in frequent traversals of the complete queue. As a consequence, this gives small sites more opportunities to gain additional resources from the larger site



(a) KTH (b) CTC

Fig. 4. Continuous reconfiguration of both KTH and CTC sites during workload processing

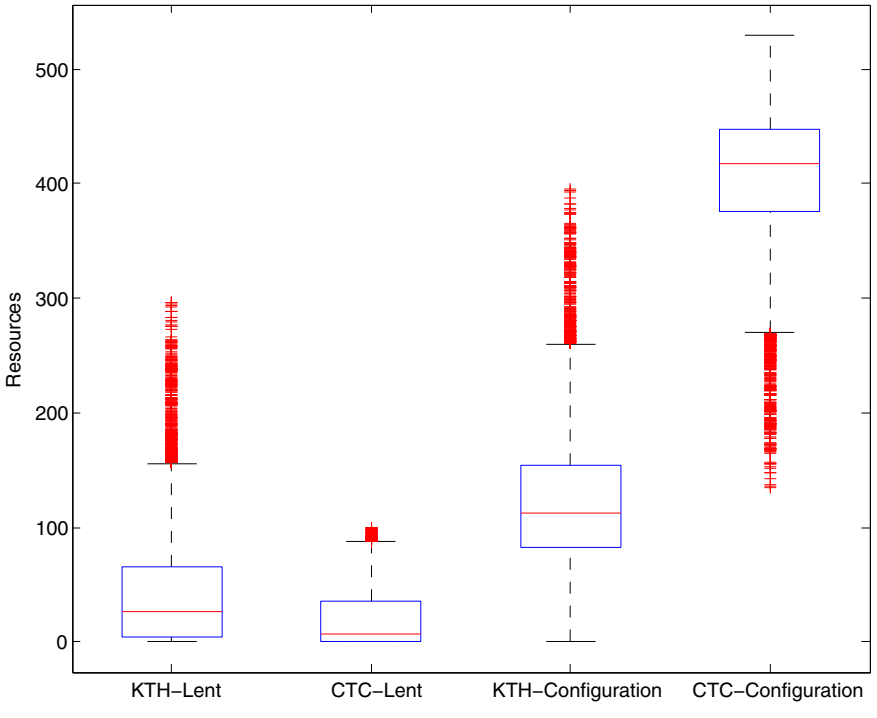


Fig. 5. Aggregated site reconfiguration properties for Setup 1. The two leftmost plots refer to leased resources by KTH and CTC to the respective partner site. The two rightmost plots state on each site's size configurations during cooperation.

to execute long waiting jobs. The opposite is not necessarily true, as the resource capacity of a small site restricts the larger site's chances.

7 Conclusion and Future Work

In this work we approached the topic of collaboration in distributed computing infrastructures from a new and more operator-centric perspective: the delegation of resources between partner sites. In order to adapt to fluctuating local user demand while constantly offering high service quality, cooperating HPC providers are enabled to mutually lease resources from other partners or grant them to him. To this end, we devised a delegation layer above the local management layer of each site and two delegation policies which combine negotiation capabilities with scheduling decisions making. This ensures both independent acting sites in a decentralized scenario as well as a situation-aware delegation behavior while leaving the local management systems largely untouched.

For evaluating the proposed collaboration scenario, we investigated several two-site setups consisting of different-sized installation by simulatively feeding them with real-world workload data. Both delegation policies demonstrated their potential realizing an enormous increase in service quality for almost all participating sites, with less robustness of second delegation approach against extremal differences in site size, leading to degradation of service quality in specific cases.

Moreover, we were able to show the dynamics of site reconfiguration in the proposed scenario: The sites frequently changed their configuration in order to fit their workload. In fact, the fluctuations in site configurations ranged from completely granting all resources to leases that multiply the site's own size. This impressively demonstrates the hidden potentials of collaboration in DCIs and should motivate operators to provide locally idle resources in order to benefit from cooperation in terms of service quality and effective resource usage.

Our next steps will be twofold: On the one hand, we will consider several restrictions in our current model: Since in practice multi-site execution of jobs might be prohibited, more powerful heuristics should yield good schedules without spreading jobs among site boundaries. This is, the have to decide between either local or remote execution.

On the other hand, advanced heuristics should be applicable under limited information exchange. To this end, they should favor a collaborative and/or partner-specific adaptive behavior. In this context—besides scalarization issues—global balancing effects and benefits of the second level partner site selection strategies have to be evaluated in larger scenarios with multiple participating sites. Those policies can possibly base on load balancing between multiple partners or cost models for resource delegation.

Finally, non job-specific leases have to be considered, allowing to "borrow" a certain amount of resources for a certain timeframe and thus to fully take care of capacity planning on these resources, as currently delivered by modern IaaS environments.

References

1. Foster, I., Kesselman, C.: Globus: A Toolkit-Based Grid Architecture. In: The Grid: Blueprint for a Future Computing Infrastructure, 1st edn., pp. 259–278. Morgan Kaufman, San Mateo (1998)
2. Gagliardi, F., Jones, B., Grey, F., Begin, M.E., Heikkurinen, M.: Building an infrastructure for scientific grid computing: status and goals of the egee project. Philosophical Transactions. Series A, Mathematical, Physical, and Engineering Sciences 363(1833), 1729–1742 (2005)
3. Ernemann, C., Hamscher, V., Schwiegelshohn, U., Streit, A., Yahyapour, R.: On advantages of grid computing for parallel job scheduling. In: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID 2002), pp. 39–46. IEEE Press, Los Alamitos (2002)
4. Kurowski, K., Nabrzyski, J., Oleksiak, A., Weglarz, J.: Scheduling Jobs on the Grid - Multicriteria Approach. Computational Methods in Science and Technology 12(2), 123–138 (2006)
5. England, D., Weissman, J.B.: Costs and Benefits of Load Sharing in the Computational Grid. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2004. LNCS, vol. 3277, pp. 160–175. Springer, Heidelberg (2005)
6. Grimme, C., Lepping, J., Papaspyrou, A.: Prospects of Collaboration between Compute Providers by means of Job Interchange. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2007. LNCS, vol. 4942, pp. 132–151. Springer, Heidelberg (2008)
7. Fölling, A., Grimme, C., Lepping, J., Papaspyrou, A.: Decentralized Grid Scheduling with Evolutionary Fuzzy Systems. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2009. LNCS, vol. 5798, pp. 16–36. Springer, Heidelberg (2009)
8. Fölling, A., Grimme, C., Lepping, J., Papaspyrou, A., Schwiegelshohn, U.: Competitive co-evolutionary learning of fuzzy systems for job exchange in computational grids. Evolutionary Computation 17(4), 545–560 (2009)
9. Kota, S.R., Shekhar, C., Kokkula, A., Toshniwal, D., Kartikeyan, M.V., Joshi, R.C.: Parameterized module scheduling algorithm for reconfigurable computing systems. In: ADCOM 2007: Proceedings of the 15th International Conference on Advanced Computing and Communications, Washington, DC, USA, pp. 473–478. IEEE Computer Society, Los Alamitos (2007)
10. Subramaniyan, R., Troxel, I., George, A.D., Smith, M.: Simulative analysis of dynamic scheduling heuristics for reconfigurable computing of parallel applications. In: Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays, pp. 230–230. ACM, New York (2006)
11. Ernemann, C., Hamscher, V., Schwiegelshohn, U., Streit, A., Yahyapour, R.: Enhanced algorithms for multi-site scheduling. In: Parashar, M. (ed.) GRID 2002. LNCS, vol. 2536, pp. 219–231. Springer, Heidelberg (2002)
12. Zhang, W., Cheng, A.M., Hu, M.: Multisite co-allocation algorithms for computational grid. In: International Parallel and Distributed Processing Symposium. IEEE Computer Society, Los Alamitos (2006)
13. Iosup, A., Tannenbaum, T., Farrellee, M., Epema, D., Livny, M.: Inter-operating grids through delegated matchmaking. Scientific Programming 16(2-3), 233–253 (2008)
14. Weissman, J.B., Grimshaw, A.S.: A federated model for scheduling in wide-area systems. In: Fifths IEEE International Symposium on High-Performace Distributed Computing (HPDC-5 1996), pp. 542–550. IEEE Computer Society, Los Alamitos (1996)

15. Feitelson, D.G., Nitzberg, B.: Job characteristics of a production parallel scientific workload on the NASA ames iPSC/860. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1995 and JSSPP 1995. LNCS, vol. 949, pp. 337–360. Springer, Heidelberg (1995)
16. Sabin, G., Kettimuthu, R., Rajan, A., Sadayappan, P.: Scheduling of parallel jobs in a heterogeneous multi-site environment. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2003. LNCS, vol. 2862, pp. 87–104. Springer, Heidelberg (2003)
17. Iosup, A., Dumitrescu, C., Epema, D., Li, H., Wolters, L.: How are Real Grids Used? The Analysis of Four Grid Traces and Its Implications. In: Gannon, D., Badia, R.M., Buyya, R. (eds.) Proceedings of the 7th IEEE/ACM International Conference on Grid Computing, pp. 262–269. IEEE Press, Los Alamitos (2006)
18. Feitelson, D.G., Weil, A.M.: Utilization and Predictability in Scheduling the IBM SP2 with Backfilling. In: Proceedings of the 12th International Parallel Processing Symposium and the 9th Symposium on Parallel and Distributed Processing, pp. 542–547. IEEE Computer Society Press, Los Alamitos (1998)
19. Schwiegelshohn, U., Yahyapour, R.: Fairness in parallel job scheduling. Journal of Scheduling 3(5), 297–320 (2000)
20. Ernemann, C., Hamscher, V., Yahyapour, R.: Benefits of global grid computing for job scheduling. In: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID 2004), pp. 374–379. IEEE Computer Society, Los Alamitos (2004)

A Moldable Online Scheduling Algorithm and Its Application to Parallel Short Sequence Mapping*

Erik Saule, Doruk Bozdağ, and Umit V. Catalyurek

Department of Biomedical Informatics, The Ohio State University
{esaule,bozdagd,umit}@bmi.osu.edu

Abstract. A crucial step in DNA sequence analysis is mapping short sequences generated by next-generation instruments to a reference genome. In this paper, we focus on efficient online scheduling of multi-user parallel short sequence mapping queries on a multiprocessor system. With the availability of parallel execution models, the problem at hand becomes a moldable task scheduling problem where the number of processors needed to execute a task is determined by the scheduler. We propose an online scheduling algorithm to minimize the stretch of the tasks in the system. This metric provides improved fairness to small tasks compared to flow time metric and suits well to the nature of the problem. Experimental evaluation on two workload scenarios indicate that the algorithm results in significantly smaller stretch compared to a recent algorithm and it is more fair to small sized tasks.

1 Introduction

The rate of increase in DNA sequence information have greatly exceeded the expectations due to the emergence of next-generation sequencing instruments, including Roche's (454) GS FLX Genome Analyzer, Illumina's Solexa IG sequencer, and Applied Biosystem's SOLiD system, which are capable of sequencing more than one billion bases a day. The massive volumes of generated data pose new computational and analytical challenges that need to be addressed rapidly to keep up with the pace of the advancements in sequencing technology.

In many genome-wide and targeted studies, such as whole-genome resequencing, transcriptome analysis, small RNA analysis, targeted sequencing, DNA methylation and ChIP sequencing, one of the first steps to analyze the generated data sequences (reads) is to map them to a reference genome. This computationally intensive process involves mapping hundreds of millions of short reads generated in a typical run of a high throughput sequencing system to a reference genome that consists of up to three and a half billion bases. Since next

* This work was supported in parts by the U.S. DOE SciDAC Institute Grant DE-FC02-06ER2775; by the U.S. National Science Foundation under Grants CNS-0643969, OCI-0904809, OCI-0904802 and CNS-0403342; and an allocation of computing time from the Ohio Supercomputer Center.

E. Frachtenberg and U. Schwiegelshohn (Eds.): JSSPP 2010, LNCS 6253, pp. 93–109, 2010.

generation sequencing instruments usually generate reads as short as 35-50 base pairs, more specialized mapping algorithms such as MapReads [1], RMAP [2], MAQ [3], SOAPv2 [4] or Bowtie [5], have been introduced and shown to be more efficient than traditional local alignment algorithms BLAST, FASTA and their variants [6,7,8] for this particular problem. Even with these new algorithms, however, the mapping process takes days on a single computer which becomes a bottleneck in the application workflow given that the goal is to be able to sequence the entire human genome in 15 minutes by the year 2013 [9]. As a natural step to speed up the mapping process, several parallelization techniques have been proposed in our recent work [10] which apply to many short sequence mapping algorithms, i.e., those based on hashing or indexing the reference genome.

In this work, we consider online scheduling of multiple parallel short sequence mapping tasks in a multi-user environment. The methods introduced in [10] describe several ways of distributing the reads and the genome data onto the processors of a cluster to parallelize short sequence mapping process. Furthermore for each method, a cost model is provided to estimate the parallel execution time for a given number of reads and a given reference genome size. Using these cost models, it is possible to determine the best parallelization method and the estimated execution time for each short sequence mapping task on a given number of processors. Therefore, in the considered scheduling problem, the number of processors to be used for executing a task is decided by the scheduler based on the current load and availability in the system. In scheduling literature, such tasks are said to be *moldable* parallel tasks[1] as opposed to *rigid* parallel tasks which require the number of processors a task will use to be provided by the user.

In this paper, we propose an algorithm to schedule moldable tasks that arise in parallel short sequence mapping. Due to the large variety of task sizes and the availability of accurate execution time estimates, we focus on minimizing the *stretch* of the tasks in the system which is defined as the time a task spends in the system normalized by its execution time. Compared to the commonly used *flow time (average turnaround time)* metric, stretch provides fairness to all tasks in the system by including the execution time of the tasks in its definition. This objective was first studied for sequential tasks without preemption [13] and later with preemption [14] in the context of bag-of-tasks applications. To the best of the authors knowledge, this work is the first to consider the minimization of the stretch objective in moldable task scheduling without preemption.

The rest of this paper is organized as follows. In Section 2, we provide background information about parallel short sequence mapping. Sections 3 and 4 present moldable task scheduling and a brief discussion of two recent studies. We give details of the proposed scheduling algorithm for moldable tasks in Section 5. Then, we report results from our experimental studies in Section 6 and conclude in Section 7.

[1] They were originally called malleable tasks [11]. Feitelson *et al.* [12] made the distinction between constant number of processors and variable number of processors by using moldable for the former case and malleable for the latter one. However, they were still called malleable in more recent works.

2 Parallel Short Sequence Mapping

The short sequence mapping problem asks for identifying the matching locations, possibly with some mismatches, of short input sequences (reads) on a reference genome. There are many mapping algorithms in the literature [1,5,3,4,2], most of which use a hash or an index table to store all consecutive same sized sub-sequences of either the reference genome or the query sequences to increase efficiency of the mapping process. The use of such data structures (i.e. hash or index), makes the problem less data dependent and enables accurate estimation of execution times by only taking global properties of the input problem into account.

A hashing based short sequence mapping algorithm consists of two major steps [10]. In the first step, a hash table is constructed by computing a hash value for each sub-sequence of the reference genome having length equal to read length. The execution time of this step can be modeled as $c_g G$, where c_g is the time needed to compute a hash value for a single sub-sequence and G is the size of the reference genome. In the second step, reads are matched to the genome by looking up their corresponding hash values in the hash table. When a fixed sized hash table is used, average number of collisions during table look-up depends on the number of entries in the table, which is proportional to genome size G. Therefore, the time required to process all reads can be modeled as $(c_r + c_c G)R$, where R is the number of reads, c_r is the constant work needed to process a single read, and c_c is a constant to capture additional work to resolve collisions. Then, the total execution time can be modeled as: $c_g G + (c_r + c_c G)R$.

As discussed in [10], straightforward methods to parallelize a mapping algorithm is to partition the reads and/or the reference genome to the processors of a cluster. This way, parallel execution time would be expressed as follows:

$$c_g \frac{G}{N_g} + (c_r + c_c \frac{G}{N_g}) \frac{R}{N_r} \tag{1}$$

where N_g and N_r respectively are the number of parts the genome and the reads are divided. For a cluster with m processors, $N_g \times N_r \leq m$ should be satisfied.

In addition to partitioning the reads and the genome, a new technique to assign reads and the genome to the processors is also introduced in [10]. In this method, called *Suffix Based Assignment (SBA)*, each processor is assigned a set of *suffixes* and is only held responsible for matching reads to the genome sequences that end in those suffixes. Each suffix consists of one or more nucleotide symbols. For example, if a processor is assigned the suffix AC, it is only responsible for matching reads that end in AC (e.g. ACCGTTA**AC**) to the genome sequences that also end in AC. Although SBA allows better parallelism, it comes with the cost of extra scan operations to compare sequences against the suffixes assigned to each processor. We represent the cost of checking the suffix of a genome and a read sequence by c_{gs} and c_{rs}, respectively. Moreover, we use N_s to denote the number of suffix groups to be considered. An example of suffix groups for $N_s = 2$ would be {A, C} and {G, T}. SBA can be applied

in combination with reads and genome partitioning and can be considered as a new dimension for parallelism. Then, under perfect load balance, the parallel execution time can be formulated as follows

$$c_{gs}\frac{G}{N_g} + c_g\frac{G}{N_gN_s} + c_{rs}\frac{R}{N_r} + (c_r + c_c\frac{G}{N_gN_s})\frac{R}{N_rN_s} \quad (2)$$

where $N_g \times N_r \times N_s \leq m$ and $N_s > 1$. Remark that if $N_s = 1$, there is no need to do suffix checking, hence with $c_{gs} = 0$ and $c_{rs} = 0$ Equation (2) reduces to Equation (1).

Please note that, by using tree-based one-to-all data distribution scheme, data distribution time, which includes distribution of input sequences and possibly reference genome to the processors of the parallel machine, becomes negligible in comparison to actual mapping computations. Therefore, it is omitted in these formulas. Furthermore, our earlier work [10] shows that these estimates are accurate.

3 Moldable Task Scheduling

3.1 Problem Formulation and Properties

In this section we discuss details about online scheduling of parallel short sequencing mapping tasks in a multi-user environment. We consider a typical online setting, where n independent tasks are dynamically submitted to a cluster of m identical processors. Arrival time of task i to the system is denoted by r_i. We use the notation $p_{i,j}$ to represent the execution time of task i on j processors. Information about arrival or execution time of the tasks are not available to the scheduler until submission. The scheduling problem we consider is to decide the number of processors π_i to be allocated for each task i and the time σ_i when the execution of task will start on the system. The completion time C_i of task i is $C_i = \sigma_i + p_{i,\pi_i}$.

In the short sequence mapping problem considered in this paper, each task i corresponds to a mapping request of R_i reads on a genome of size G_i. Therefore, for each task i and each number of processors $j \leq m$, $p_{i,j}$ is computed using Equation (2) by replacing G with G_i and R with R_i. The values for N_r, N_g and N_s are chosen such that the total execution time predicted by this equation is minimized for the given values of G_i, R_i and j. In short sequence mapping, storing the genome in a hash table implies high memory requirement that prevents two tasks to be executed simultaneously on the same processor. Thus, preemption is not allowed. *Monotony* of computation time and absence of super-linear speedup are common assumptions in moldable scheduling. One can check that they are valid for the parallel short sequence mapping tasks.

3.2 Objective Functions

The general objective in online scheduling is to execute all submitted tasks without delaying their execution too much in the system. A desired property of a scheduler is to avoid starvation while ensuring an overall good response time.

The most studied objective functions in moldable task scheduling are based on aggregation of completion time (makespan) of the tasks, such as the minimization of maximum completion time and minimization of average completion time. A common technique to optimize completion time is to use dual-approximation [15,16]. This technique consists of choosing a target value for the objective and then to decide the most efficient number of processors a task should use to finish before the targeted completion time. Such number of processors is commonly called the canonical number of processors. A major disadvantage of using completion time in the objective function is the requirement of a time origin, which does not suit well to online scheduling problems.

A commonly used metric in online scheduling that does not require a time origin is the *flow time* (also known as turnaround time). Flow time of a task i is the time the task spends in the system and is calculated as $F_i = C_i - r_i$. Two related objective functions are the minimization of maximum flow time ($F_{max} = \max_i F_i$) and the minimization of the average flow time ($\frac{\sum_i F_i}{n}$). The former is especially well known for preventing starvation and is usually optimized by using the first-come first-serve (FCFS) ordering. Examples of flow minimization can be found in [17,18,19].

Since the flow time metric does not take the size of the tasks into account, objective functions that utilize this metric tend to create schedules in which small tasks spend as much time in the system as the large tasks. This results in small tasks waiting in the system queue longer than the large tasks, hence introduces unfairness against small tasks. To avoid this situation, the *stretch* metric can be used to replace flow time in the objective function. The stretch of a task is defined as the time spent by the task in the system normalized by its processing time. This metric has been studied for online scheduling of rigid tasks with preemption in [20] and for sequential task scheduling without preemption in [13] and then with preemption in [14]. However, to the best of the authors' knowledge, it has never been used nor defined in online moldable scheduling without preemption. We use the processing time of the task on one processor for normalization and the stretch of a task i is $s_i = \frac{C_i - r_i}{p_{i,1}}$. Corresponding objective functions for stretch are the minimization of maximum stretch ($S = \max_i s_i$) and the average stretch ($\frac{\sum s_i}{n}$).

Using an adversary technique, one can prove that in an online setting it is not possible to get an approximation algorithm for the minimization of maximum or average stretch objectives without preemption even if the system is composed of a single processor. Adversary technique works by dynamically constructing the instance that worsens the performance the most by taking advantage of the decisions of the algorithm. In this case, the idea is to first feed one long task to the scheduler. Once the execution of that task starts, the adversary submits a bunch of much smaller tasks to the system. Since these small tasks cannot start before the execution of the first task completes, the stretches of the small tasks are as large as the ratio between the execution time of the smallest task to that of the largest task. Such analysis is usually only useful for designing

approximation algorithms. However, Section 6 indicates that similar effects also appear in practice.

In the job scheduling literature, objective functions similar to stretch have also been used. A commonly used objective function is the slowdown of a rigid task, which is the time the task spends in the system divided by its processing time. Since the task is rigid, the processing time is the actual execution time of the task. As a result, the slowdown always greater than one. Stretch can be considered as an extension of slowdown for the moldable tasks model.

A variant of the slowdown objective function is Bounded slowdown (BSLD) [12], which is used to avoid over-emphasizing the significance of small tasks. In this objective function, the processing time of the tasks are assumed to be greater than a given constant. Since this may resut in some taks to have a slowdown less than 1, the slowdown values between 0 and 1 are rounded up to 1. Due to the rounding, BSLD is not appropriate for the moldable task model, as the stretch or slowdown values of interest for this model can be less than 1.

Another closely related objective function is the Xfactor [21], which is defined as $\frac{queuingtime+p_{i,1}}{p_{i,1}}$. Xfactor is always greater than one and it does not take into account the number of processors used to execute a task.

3.3 Backfilling Strategies

In most scheduling algorithms, tasks are scheduled as soon as possible in the order of their arrival times. This approach tends to create holes in the schedule, which can later be utilized using a conservative or an aggressive backfilling strategy. In conservative backfilling, a task is scheduled in the first hole that can accommodate the task. If no such hole exists, the task is scheduled at the end of the schedule. In aggressive backfilling, a task is scheduled in the first hole that has enough number of available processors. If this creates a conflict, the task in conflict that has the largest start time is rescheduled. This approach provides a much better utilization of the cluster by reducing the number and size of the holes in the schedule. However, it tends to reschedule large tasks several times, causing longer delays for them. More details on backfilling strategies can be found in [22].

4 Analysis of Existing Solutions

4.1 The Fair-Share Scheduling Algorithm

The fair-share scheduling algorithm has been proposed in [19] and refined in [18] to optimize the average turnaround time. The basic principle of the original algorithm is to greedily schedule tasks one by one to minimize their completion time using aggressive backfilling. However, this approach leads to scheduling each task to execute in parallel using all processors in the system. Since efficiency usually decreases with the number of processors, tasks spend too much time in the system using this approach. To avoid such scenarios, the fair-share algorithm limits the maximum number of processors that can be allocated to each task.

This limit is called the fair-share limit, and finding a good value for the fair-share limit is the motivation behind the mentioned studies. The fair-share limit of a task i was first set to the ratio of work associated with the task to the total work associated with all tasks pending in the system. Using this limit is stated to be fair since it allocates more processors to larger tasks while limiting the maximum allocation by the weight of the tasks. It was shown that using a fair-share limit of $\frac{\sqrt{p_{i,1}}}{\sum_k \sqrt{p_{k,1}}}$ leads to better results. However, this value was reported to be too restrictive and multiplied by an overbooking factor to allow the scheduler to consider a larger number of possibilities [18].

The fair-share algorithm induces starvation due to aggressive backfilling which can delay all the tasks but the first to be executed. Therefore, the tasks are partitioned in multiple queues based on the their sizes. Ensuring that the first task of each queue is never delayed reduces starvation. To further reduce starvation, the Xfactor of a task is introduced: $\text{Xfactor}(i) = \frac{t+p_{i,1}-r_i}{p_{i,1}}$, where t is the current time. A task whose Xfactor exceeds a certain threshold is no longer allowed to be rescheduled by the aggressive backfilling technique.

4.2 Iterative Moldable Scheduling Algorithm

The fair-share algorithm provides fairly good performance but requires tuning many parameters. In [18], Sabin *et al.* proposed a parameter-free iterative scheduling technique which is reported to outperform the fair-share algorithm and its variants.

The fundamental idea in the algorithm is to make all tasks rigid, i.e., decide the number of processors to be allocated for each task. Then, the tasks are scheduled using a conservative backfilling technique. The order in which the task are considered for backfilling is not given in [18]. In the following we assume that tasks are considered in the FCFS order.

The question of how many processors to allocate for each task is addressed using a simple principle. The algorithm starts by allocating one processor to each task and computing the corresponding schedule. Then, the task that would have the most reduction in its processing time by using an extra processor is found. Subsequently, an additional processor is assigned to that task and a new schedule is computed. If the new schedule has a better average turnaround time, then the extra processor allocation is confirmed and the process is repeated iteratively. Otherwise, the algorithm rolls back to the previous allocation state and never tries to assign an additional processor to this task again.

The algorithm implicitly assumes that the processing time of a task strictly decreases with the number of processors. However, this assumption may not hold in practice. For example, it is fairly common that parallel algorithms require a number of processors which is a power of two. Similarly, in the short sequence mapping problem the values of N_r, N_g and N_s in Equation (2) have to be integer. If the number of processors m is prime, it is likely that the optimal partitioning scheme uses at most $m-1$ processors, which induces steps in the speedup function.

Improvements to the algorithm: Existence of steps in the speedup function results in early termination of the iterative scheme in the algorithm of Sabin *et al.* [18]. To remedy this situation, we propose the following modification. If task i is allocated x processors, instead of considering its execution on $x+1$ processors, we consider its execution on $x+k$ processors ($k \geq 1$) such that $\frac{p_{i,x}-p_{i,x+k}}{k}$ is maximal. If the speedup function is convex, this modification behaves the same as the original algorithm. If the speed up function is not convex, the modification allows to skip the allocation sizes that would lead to low efficiency (and thus skips steps). Throughout the paper, we refer to this variant of the algorithm as the *improved iterative* algorithm.

5 Deadline Based Online Scheduling

In this work, we propose an algorithm called *Deadline Based Online Scheduling (DBOS)* with the goal of minimizing the stretch of the tasks in the system. Throughout the section, we consider a typical system where the scheduler is invoked when a task enters or exits the system. Using the `DBOS` algorithm, the scheduler computes a new schedule for all tasks pending in the system queue. Tasks that have already started execution are kept running.

The outline of the `DBOS` algorithm is presented in Algorithm 1. The main idea in the algorithm is to compute the "best" achievable maximum stretch, denoted by S, using a binary search within lines 2–14. At each iteration of the binary search, a new schedule is computed by calling the `MoldableEDF` (for Moldable Earliest Deadline First) procedure using the current value of S. If the returned schedule is not feasible, S is increased. Otherwise it is decreased to find a tighter bound for maximum stretch. Since there is no apriori upper bound on S, the algorithm starts with computing one in lines 2–6.

Once the "best" feasible value of S is found, it is multiplied by an online factor ρ. The reason for relaxing the S value is to increase the efficiency of the system as well as to leave potentially more processors to the tasks that will arrive in the future. Furthermore, this helps improving the performance in the adversary scenario discussed in Section 3.2. The online factor ρ is the key to the online aspect of the `DBOS` algorithm.

In lines 20–30 of Algorithm 1, the details of the `MoldableEDF` procedure is given. Given a value of S, `MoldableEDF` starts by computing a deadline $D_i = r_i + p_{i,1}S$ for each task i (lines 21–22). This reduces the problem to scheduling the tasks before their deadlines. Then, the tasks are scheduled greedily in non-decreasing order of their deadlines. For each task i, the smallest number of processors j that allows the task to finish before its deadline D_i without moving any previously scheduled task is determined. Finally, task i is scheduled to start as soon as possible on j processors. If it is not possible to schedule task i before D_i, the constructed schedule is labeled as infeasible. Remark that the core of the deadline scheduling algorithm from line 23 to line 29 is generic. It could be used for a classical scheduling problem of moldable tasks with deadline.

The algorithm has several interesting properties. First of all, if `MoldableEDF` was an exact algorithm, then the optimal maximum stretch would be found.

Algorithm 1. Deadline Based Online Scheduling Algorithm

```
1: procedure DBOS(INPUT: ρ, OUTPUT: π*, σ*)
2:     LB ← 0, S ← 1
3:     while Not Feasible (π, σ) do ▷ Compute an initial feasible maximum stretch
4:         S ← 2S
5:         (π, σ) ← MoldableEDF(S)
6:     UB ← S
7:     while UB ≠ LB do ▷ Find the best maximum stretch using a binary search
8:         S ← (UB+LB)/2
9:         (π, σ) ← MoldableEDF(S)
10:        if Feasible (π, σ) then
11:            (π*, σ*) ← (π, σ)
12:            UB ← S
13:        else
14:            LB ← S
15:    (π^ρ, σ^ρ) ← MoldableEDF(ρS)      ▷ Relax S by a factor of ρ if it is feasible
16:    if Feasible (π^ρ, σ^ρ) then
17:        (π*, σ*) ← (π^ρ, σ^ρ)
18:    return (π*, σ*)
19:
20: procedure MoldableEDF(S)
21:    for all i ≤ n do                  ▷ Compute a deadline for each task
22:        D_i ← r_i + p_{i,1}S
23:    Construct initial processor allocation using information about running tasks
24:    for all task i in non-decreasing D_i order do
25:        for all j from 1 to m do
26:            x ← earliest time that j processors are available for p_{i,j} units of time
27:            if x + p_{i,j} ≤ D_i then
28:                π_i ← j; σ_i ← x
29:                Exit inner for loop
30:    return (π, σ)
```

The deadline scheduling problem as well as the maximum stretch optimization problem are NP-Complete [23]. However, it is likely that if an approximation algorithm for the deadline scheduling problem was known, it would lead to an approximation algorithm for the maximum stretch optimization problem.

`MoldableEDF` is a greedy algorithm and is not optimal as it can fail to find the best feasible solution. However, the `MoldableEDF` is based on two principles that make it efficient. First, the tasks are considered in non-decreasing order of deadlines. This principle, called Earliest Deadline First, leads to optimality in single processor deadline scheduling problems and provides guaranteed approximation for the sequential task scheduling problem on an arbitrary number of processors. Second, the algorithm allocates the minimum number of processors that ensures a task matches its deadline. This decision maximizes the processor availability for the other tasks in the system, hence helps keeping the system efficiency high. Moreover, it helps avoiding local optima due to presence of steps in the speedup

Table 1. (Left) Sequencing machines and the number of reads each of them produces in a single run. (Right) Genomes and their sizes.

Sequencing machine	Number of reads
454 GS FLX Genome Analyzer	1 million
Solexa IG sequencer	200 million
SOLiD system	400 million

Genome	Size (bases)
E. Coli	4.6 million
Yeast	15 million
A. Thaliana	100 million
Mosquito	280 million
Rice	465 million
Chicken	1.2 billion
Human	3.4 billion

function. This principle is similar to the canonical number of processors used in makespan optimization.

6 Experiments

Execution time of short sequence mapping tasks vary significantly depending on the size of the reference genome and the number of reads to be mapped (see Table 1). For instance, a targeted sequence analysis involves mapping a few million reads to a genome segment of a few hundred thousand bases and can be carried out in a couple of minutes. On the other hand, a whole-genome resequencing application requires mapping hundreds of millions of reads and may take a few hours for mosquito and a few days for human genome. In this section, we report on the simulation results of the `DBOS` algorithm on a 512-processor cluster using two workload scenarios that reflect such variety in task execution times. The first scenario is based on a log file from a supercomputing center and is included to assess the performance of the algorithm on well known data. The second scenario is designed to simultate the load of a cluster dedicated for short sequence mapping tasks.

In the first scenario, we used a real log file (SDSC Par 96 in [24]) of parallel jobs submitted to the San Diego Supercomputing Center (SDSC). This file contains information about a task's arrival time, runtime on the system and the number of processors used for its execution. We considered the first $5,000$ tasks, and similar to [18], we used the Downey model [25] to estimate the scalability of the tasks. The Downey model requires two parameters for each task: maximum parallelism and variance of parallelism of the task. The value of the maximum parallelism is randomly selected between p and 512, where p is the recorded number of processors used to execute the task in the log file. The value of the variance of parallelism is randomly selected between 0 and 2 which is a realistic range for this parameter [25]. Since the Downey model is stochastic, 10 different instances were generated.

In the second scenario, each workload consists of 5,000 parallel short sequence mapping tasks and each task arrives at the cluster with an inter-arrival time chosen from a exponential distribution of parameter λ_i. We varied λ_i to obtain 6 different load conditions, where load is defined as the ratio of the sum of

sequential processing times of all tasks to the time that elapsed between the arrival of the first and the last task. In other words, for a load of l, if all tasks were executed sequentially, then total computing power of l processors would be used to execute the tasks over the time for which the activity on the cluster is simulated. Therefore, in our tests, if the load is larger than 512, the cluster is clearly overloaded. However, due to non-linear scalability of the tasks and random arrival times, it is very likely that the cluster gets overloaded even for load values less than 512. A task in this scenario represents a mapping operation of short sequences generated by one of the sequencing machines to one of the genomes listed in Table 1. The sequencing machine and the genome associated with a task is chosen randomly and the parallel execution time of the task is computed using the formulas from Section 2. The sequential processing time of the generated tasks vary between 30 seconds and 22 days.

Results of the DBOS algorithm are presented in comparison to the iterative algorithm of Sabin *et al.* [18] which was described in Section 4.2.

6.1 Downey Model

First we present aggregate results from 10 runs using Downey model on the SDSC log file. Since 5,000 tasks are scheduled in each run, we had scheduling information about 50,000 tasks in 10 runs. In Figure 1(a), the flow time of these 50,000 tasks are shown in increasing flow time order for DBOS and the iterative algorithms (the improved version of the iterative algorithm is not presented here as it is equivalant to the original iterative algorithm since there are no steps in speedup functions of the Downey model). Figure 1(b) shows the corresponding chart with stretch on the y-axis. Due to wide variation of flow times and stretch values, log scale is used in the y-axis of both charts. These results show that on the average DBOS provides a better flow time and stretch compared to the iterative algorithm. Recall that if a task has a stretch greater than 1, it means that the time it spends in the system is greater than its sequential execution time. In other words, the speedup gain due to parallel execution is lost. The iterative algorithm resulted in more than 23% of the tasks to have stretch greater than 1, whereas the corresponding quantity was only 6% for DBOS with $\rho = 1$. The results improved even further when the value of ρ is increased to 1.5. In that case, only 1% of the tasks had a stretch greater than 1. In Figure 1(c), the percentage of tasks with stretch greater than 1 is shown for different task-size groups. The results indicate that the iterative algorithm results in a relatively unfair schedule by penalizing smaller tasks more in terms of their stretch. For example, 34% of the tasks in the smallest task-size group have a stretch larger than 1. DBOS results in a more fair schedule, where less than 7% and 1% of the tasks had a stretch greater than 1 even for the smallest tasks with $\rho = 1$ and $\rho = 1.5$, respectively.

Note that larger tasks can afford longer delays without much degradation in their stretch. However, smaller jobs suffer more especially when the cluster is overloaded. This is similar to the worst case online scenario on a single processor as mentioned in Section 3.2, in which a very short task arrives just after a

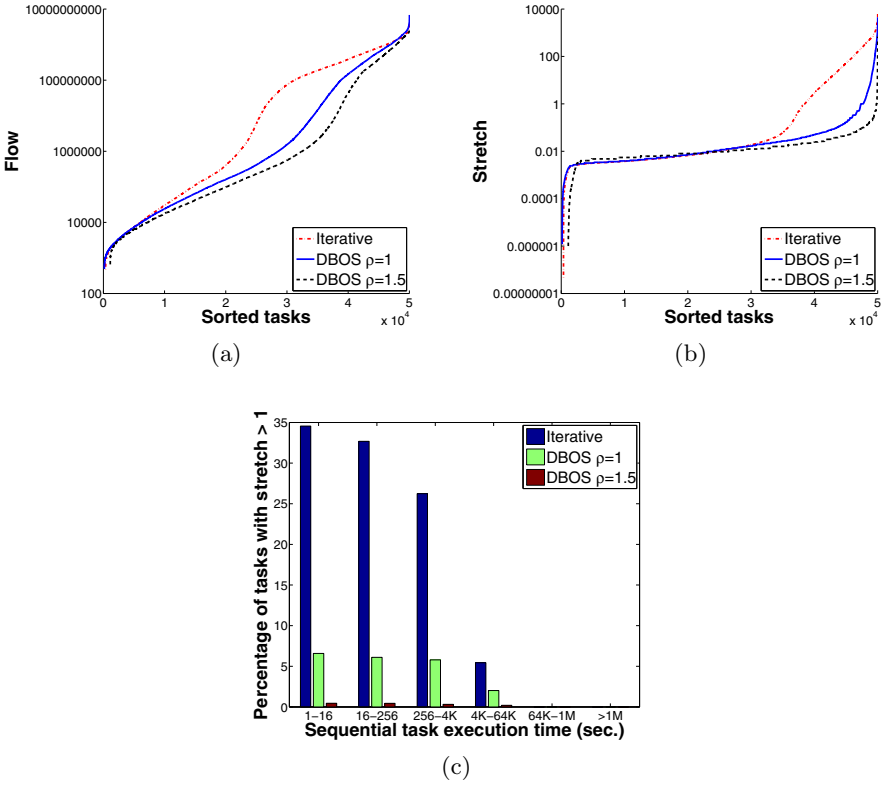


Fig. 1. Comparison of DBOS and the iterative algorithm on the SDSC/Downey workload. The y-axis is in log scale for (a) and (b). The lower is the better in all figures.

very long task is scheduled. Existence of some tasks getting a stretch over 100 in our experiments is the proof that such phenomenon appears also in practice. Nevertheless, if a value larger than 1 is used for the online factor ρ this behavior occurs rarely.

6.2 The Short Sequence Mapping Application

In the second set of experiments we considered workloads consisting of short sequence mapping tasks as described in the second scenario above. We generated 6 different load cases and for each case we generated 20 workloads. Since the instance generation will not provide the same load when run with the same parameters, we considered a range of load values around a targeted load value (and tuned the λ_i parameter to reach this load). The 6 load cases in the experiments correspond to the following ranges of load values: 100-115, 200-230, 330-360, 150-500, 500-570 and 640-710. Note that in the last two cases the cluster is overloaded (as the load is greater than the number of processors). These cases are included to see the performance of the algorithm in extreme load conditions.

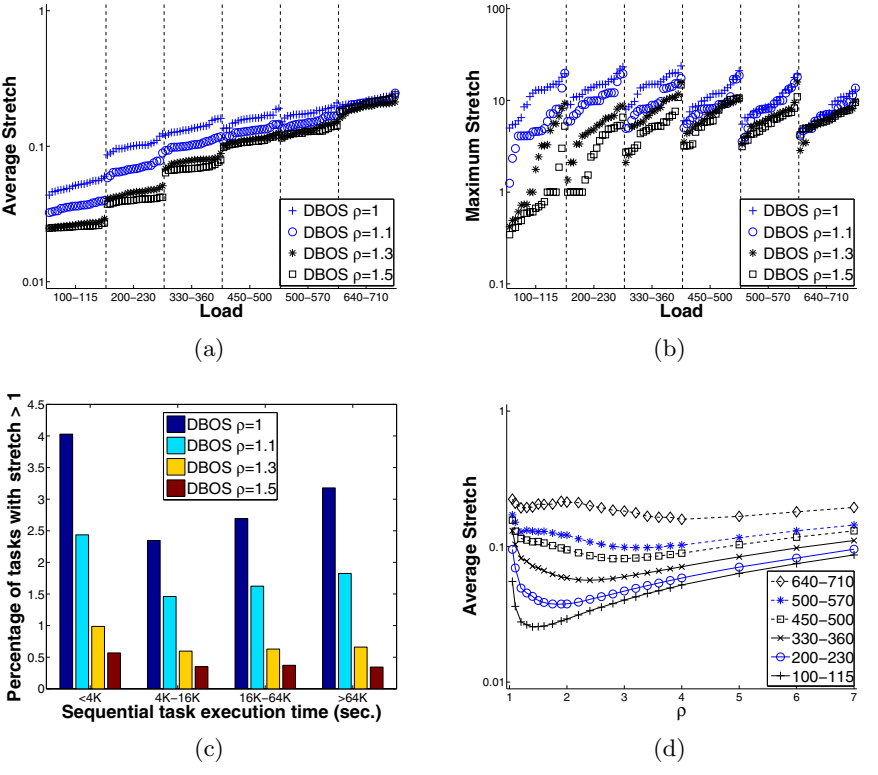


Fig. 2. Impact of the online factor ρ on the performance. The y-axis is in log scale for (a), (b) and (d). The lower is the better in all figures.

We started with assessing the impact of the online factor ρ of the DBOS algorithm and used values of ρ chosen from the set $\{1, 1.1, 1.3, 1.5\}$. Recall that the parameter ρ allows to take the online characteristics of the problem into account by relaxing the instant maximal stretch to improve overall efficiency. In Figure 2(a), average stretch achieved by the DBOS algorithm with different ρ values are given under different load cases. For each load case and for each ρ value, the average stretch values are shown sorted. Figure 2(b) displays the corresponding results for maximum stretch values. In Figure 2(c), the percentage of tasks with stretch greater than 1 is shown for different task-size groups using the aggregate results from all 20 workloads that have load in the 330-360 range.

The results in Figure 2 suggest that both average and maximum stretch improves significantly with ρ until $\rho = 1.3$, after which the improvement is marginal. In general, using a ρ value greater than 1 results in an increase in the stretch of the tasks with extremely small stretch and a decrease in the stretch of the tasks with extremely large stretch (results were similar to those in Figure 1(b), hence omitted). Therefore, using larger ρ values helps reducing the variance of stretch as well as the average and maximum stretch. As seen in

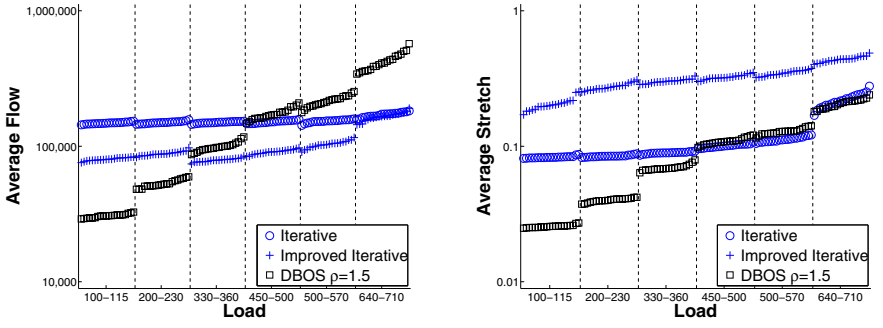


Fig. 3. Comparison of DBOS and the iterative algorithm on short sequence mapping application workloads

Figure 2(c), small sized tasks benefit the most from larger ρ values, as they are more likely to get large stretch values due to cases similar to the worst-case scenario described in Section 3.2. As the load in the system increases, there are far more tasks in the system and the online factor becomes less effective as it is no longer sufficient to keep a portion of the processors available for the tasks that will arrive in the future. Figures 2(a) and 2(b) illustrate that the online factor has very little impact in the two extreme load cases, where load is greater than 500.

In order to determine a reasonable range of values for the online parameter ρ, we computed the average stretch for different ρ values under different load conditions. The results of this experiment are given in Figure 2(d), where each point is the average over 20 instances of similar loads. Since the variance of average stretch values is low, (see Figure 2(a)), the standard deviation is omitted in this figure for clarity. The results show that the optimal value of ρ depends on the load of the system. The average stretch quickly drops when ρ increases as more room is created for small tasks. Then it slowly increases as all the tasks get delayed and some machines of the cluster are left idle. The shape of the curve allows easy estimation of the optimal ρ with a gradient method. Moreover, note that the average stretch has small variation around the optimal ρ value. For instance, for a load between 330 to 360, the optimal ρ value is 2.4 and all ρ values between 1.6 and 3.8 result in average stretch values within 20% of the optimal. Therefore, fine tuning of the ρ value is not essential as long as unreasonable values are avoided. In the rest of the experiments, the value of ρ is set to 1.5 which is a reasonable value for underloaded cluster scenarios.

In Figure 3 the results of the DBOS algorithm on short sequence mapping workloads are presented in comparison to the two variants of the iterative algorithm mentioned in Section 4.2: the original algorithm in [18], and the improved version for non-convex speedup function. These two variants lead to different results due to steps in the speedup curves of the short sequence mapping tasks.

Results in Figure 3(a) show that the improved version leads to around 50% improvement in flow time; steps in speedup curves prevent the original version from using available parallelism. On the overloaded cases, the two versions are comparable since there are more tasks in the queue and both algorithms use all available processors. If the average flow time is the target metric in an application, our proposed improvement should be used in the iterative algorithm to handle non-convex speedup curves.

However, under-utilization of the cluster in the original iterative algorithm results in better stretch values. Indeed, the improved algorithm tends to utilize all processors in the cluster, thus tasks entering the system are delayed and get large stretch values. The original version results in many processors to remain idle, therefore, tasks that enter the system are scheduled immediately and obtain small stretch values. However, note that if the application had required the number of processors to be a power of two, the original iterative algorithm would never schedule a task on more than two processors, hence would not get a stretch better than 0.5. This is worse than the improved version which reaches an average stretch of 0.3. On the other hand, if the first step in the speedup curve appears on a large number of processors, the behavior of the original iterative algorithm converges to that of the improved one.

As clearly seen in Figure 3, DBOS achieves better stretch than both variants of the iterative algorithm. The difference is especially larger for low load conditions, where more than 70% improvement is achieved relative to the original iterative algorithm. The performance of the original iterative algorithm is comparable with DBOS only under the cases where the cluster had a load greater than 400. DBOS outperforms the revision of the iterative algorithm up to 85% on low load cases.

In terms of flow time, DBOS achieves better results than the iterative algorithm under low and medium load and worse results only in overloaded cluster conditions. Results in Figure 3 leads to the conclusion that DBOS achieves a balance between inefficient over-parallelism as in the case of improved iterative algorithm, and under-utilization of the cluster as in the case of original iterative algorithm. Therefore, except for extreme load conditions, it usually gives the best stretch and flow time among the considered algorithms.

The scheduling overhead of both DBOS and the iterative algorithm are low and mainly depends on the number of tasks in the queue. On a regular desktop (2.4Ghz Intel Core2 processor, 2GB of memory), our unoptimized implementation of DBOS and the iterative algorithm take about 20 to 30 seconds to schedule 5000 tasks. Despite a greedy algorithm would deliver the schedules faster, the computation times of the benchmarked algorithms are far from being prohibitive since the execution of tasks in a cluster can last for hours and since the scheduling process does not interfere with tasks already being executed.

7 Conclusion

The most computationally demanding step in DNA sequence analysis is mapping sequences generated by next-generation sequencing instruments to a reference

genome. In this paper, we investigated online scheduling of multiple parallel short sequence mapping tasks in a multi-user environment. Availability of accurate estimates of parallel execution times of short sequence mapping queries allows using the moldable task model in the scheduling process. Existing studies mainly focus on optimizing the average flow time of tasks, which produces schedules unfair against small tasks. In the context of sequential tasks, one of the proposed solutions to address the fairness issue was the use of stretch metric. To the best of our knowledge, the work presented in this paper is the first that uses the stretch metric for moldable task scheduling without preemption. Experiments on two different workload scenarios, one based on the log of a production batch system and one reflecting realistic use-case scenario of the short sequence mapping application, showed that the proposed `DBOS` algorithm provides better schedules than the compared algorithms in terms of the stretch metric while improving the flow time on many cases. The results demonstrated that `DBOS` achieves a balance between inefficient over-parallelism and under-utilization of the cluster, two competing issues regarding online task scheduling.

References

1. Applied Biosystems, MapReads: SOLiD System Color Space Mapping Tool, `http://solidsoftwaretools.com/gf/project/mapreads/`
2. Smith, A.D., Xuan, Z., Zhang, M.Q.: Using quality scores and longer reads improves accuracy of solexa read mapping. BMC Bioinformatics 9(1), 128 (2008)
3. Li, H., Ruan, J., Durbin, R.: Mapping short dna sequencing reads and calling variants using mapping quality scores. Genome Research 18(11), 1851–1858 (2008)
4. Li, R., Yu, C., Li, Y., Lam, T.W.W., Yiu, S.M.M., Kristiansen, K., Wang, J.: SOAP2: an improved ultrafast tool for short read alignment. Bioinformatics 25(15), 1966–1967 (2009)
5. Langmead, B., Trapnell, C., Pop, M., Salzberg, S.L.: Ultrafast and memory-efficient alignment of short dna sequences to the human genome. Genome Biology 10(3), R25 (2009)
6. Altschul, S., Gish, W., Miller, W., Myers, E., Lipman, D.: Basic local alignment search tool. Journal of Molecular Biology 215, 403–410 (1990)
7. Pearson, W.R., Lipman, D.J.: Improved tools for biological sequence comparison. Proc. National Academy of Sciences 85, 2444–2448 (1988)
8. Zhang, Z., Schwartz, S., Wagner, L., Miller, W.: A greedy algorithm for aligning DNA sequences. Journal of Computational Biology 7(1/2), 203–214 (2000)
9. Davies, K.: Pacific Biosciences preparing the 15-minute genome by 2013. Bio IT World (2008)
10. Bozdag, D., Barbacioru, C.C., Catalyurek, U.: Parallel short sequence mapping for high throughput genome sequencing. In: Proc. of the International Parallel and Distributed Processing Symposium (2009)
11. Turek, J., Wolf, J.L., Yu, P.S.: Approximate algorithms scheduling parallelizable tasks. In: Proc. of the fourth Symposium on Parallel Algorithms and Architectures, pp. 323–332. ACM, New York (1992)
12. Feitelson, D.G., Rudolph, L., Schwiegelshohn, U., Sevcik, K.C., Wong, P.: Theory and practice in parallel job scheduling. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1997 and JSSPP 1997. LNCS, vol. 1291, pp. 1–34. Springer, Heidelberg (1997)

13. Bender, M., Muthukrishnan, S., Rajaraman, R.: Improved algorithms for stretch scheduling. In: Proc. of the Symposium on Discrete Algorithms, pp. 762–771 (2002)
14. Legrand, A., Su, A., Vivien, F.: Minimizing the stretch when scheduling flows of biological requests. In: Proc. of the Symposium on Parallelism in Algorithms and Architectures (2006)
15. Jansen, K., Porkolab, L.: Linear-time approximation schemes for scheduling malleable parallel tasks. In: Proc. of 10th SODA, pp. 490–498 (1999)
16. Mounie, G., Rapine, C., Trystram, D.: A 3/2-approximation algorithm for scheduling independent monotonic malleable tasks. SIAM J. Comput. 37(2), 401–412 (2007)
17. Drozdowski, M., Dell'Olmo, P.: Scheduling multiprocessor tasks for mean flow time criterion. Computers and Operations Research 27(6), 571–585 (2000)
18. Sabin, G., Lang, M., Sadayappan, P.: Moldable parallel job scheduling using job efficiency: An iterative approach. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2006. LNCS, vol. 4376, pp. 94–114. Springer, Heidelberg (2007)
19. Srinivasan, S., Subramani, V., Kettimuthu, R., Holenarsipur, P., Sadayappan, P.: Effective selection of partition sizes for moldable scheduling of parallel jobs. In: Sahni, S.K., Prasanna, V.K., Shukla, U. (eds.) HiPC 2002. LNCS, vol. 2552, pp. 174–183. Springer, Heidelberg (2002)
20. Muthukrishnan, S., Rajaraman, R., Shaheen, A., Gehrke, J.: Online scheduling to minimize average stretch. In: Proc. of FOCS, pp. 433–443 (1999)
21. Srinivasan, S., Krishnamoorthy, S., Sadayappan, P.: A robust scheduling technology for moldable scheduling of parallel jobs. In: Proc. of Cluster 2003, pp. 92–99 (2003)
22. Srinivasan, S., Kettimuthu, R., Subramani, V.: Selective reservation strategies for backfill job scheduling. In: Blaze, M. (ed.) FC 2002. LNCS, vol. 2357, pp. 55–71. Springer, Heidelberg (2003)
23. Garey, M.R., Johnson, D.S.: Computers and Intractability. Freeman, New York (1979)
24. Feitelson, D.: Parallel workloads archive, http://www.cs.huji.ac.il/labs/parallel/workload/
25. Downey, A.B.: A parallel workload model and its implications for processor allocation. Cluster Computing 1(1), 133–145 (1998)

Dynamic Proportional Share Scheduling in Hadoop

Thomas Sandholm and Kevin Lai

Social Computing Lab, Hewlett-Packard Labs, Palo Alto, CA 94304, USA
{thomas.e.sandholm,kevin.lai}@hp.com

Abstract. We present the Dynamic Priority (DP) parallel task scheduler for Hadoop. It allows users to control their allocated capacity by adjusting their spending over time. This simple mechanism allows the scheduler to make more efficient decisions about which jobs and users to prioritize and gives users the tool to optimize and customize their allocations to fit the importance and requirements of their jobs. Additionally, it gives users the incentive to scale back their jobs when demand is high, since the cost of running on a slot is then also more expensive. We envision our scheduler to be used by deadline or budget optimizing agents on behalf of users. We describe the design and implementation of the DP scheduler and experimental results. We show that our scheduler enforces service levels more accurately and also scales to more users with distinct service levels than existing schedulers.

Keywords: MapReduce, Dynamic Priority, Task Scheduling.

1 Introduction

Large compute clusters have become increasingly easier to program because of simplified parallel programming models such as MapReduce. At the same time, the costs for deploying and operating such clusters are significant enough that users have a strong incentive to share them. However, MapReduce was initially designed for small teams where resource contention can be resolved using FIFO scheduling or through social scheduling.

In this paper, we examine different task-scheduling methods for shared Hadoop (an open source implementation of MapReduce) clusters. As a result of our analysis of Hadoop scheduling, we have developed the Dynamic Priority (DP) scheduler, a novel scheduler that extends the existing FIFO and fair-share schedulers in Hadoop. This scheduler plug-in allows users to purchase and bid for capacity or quality of service levels dynamically. The capacity allotted, represented by Map and Reduce task slots, is proportional to the spending rate a user is willing to pay for a slot and inversely proportional to the aggregate spending rate of all existing users. When running a task on the alloted slot, that same spending rate is deducted from the user's budget.

This simple mechanism allows the DP scheduler to make more efficient decisions about which jobs and users to prioritize and gives users the ability to

E. Frachtenberg and U. Schwiegelshohn (Eds.): JSSPP 2010, LNCS 6253, pp. 110–131, 2010.

optimize and customize their allocations to fit the importance and requirements of their jobs. Additionally, it gives users the incentive to scale back their jobs when demand is high, since the cost of running on a slot is then also more expensive. We envision the DP scheduler to be used by deadline or budget optimizing agents on behalf of users. In comparison to existing schedulers, the DP implementation is simpler because it does not rely on heuristics, while still providing preemption and being work-conserving.

We present the design and implementation of the DP scheduler and experimental results. We show that our scheduler enforces service levels more accurately and also scales to more users with distinct service levels than existing schedulers. We also show how the dynamics of budgets and spending rates affect job completion time. The DP scheduler enables cost-driven scheduling across Hadoop clusters potentially operated from different sites and administrative domains.

This paper is organized as follows. In Section 2 we review the current Hadoop schedulers. We then describe the design and rationale behind our scheduler implementation in Section 3. In Section 4 and Section 5 we present and discuss a series of experiments used to evaluate our scheduler. Finally, we relate our work to previous work in Section 6 and conclude in Section 7.

2 Hadoop MapReduce

Apache Hadoop [1] is an open source version of the MapReduce parallel programming framework [2] and the Google Filesystem [3]. Historically it was developed for the same reasons Google developed their corresponding protocols, to index and analyze a huge number of Web pages. Data parallel programming or data-intensive scalable computing (DISC) [4] have since been deployed in a wide range of applications (e.g., OLAP, data mining, scientific computing, media processing, log analysis and data warehousing [5]). Hadoop runs on tens of thousands of nodes in production at Yahoo!, and Google uses their implementation heavily in a wide range of production services such as Google Earth [6].

The MapReduce model allows programmers to focus on designing the application workflow and how data are filtered and aggregated in the different stages of these workflows. The system takes care of common distributed systems tasks such as scheduling, input partitioning, failover, replication, and distributed sorting of intermediate results. The main benefits compared to other parallel programming models are the inherent data-local scheduling, and the ease of use, leading to increased developer productivity and application robustness.

In the seminal deployment at Google [2] the MapReduce architecture comprises one master and many workers. The input data is split and replicated in 64 MB blocks across the cluster. When a job executes, the input data is partitioned among parallel map tasks and assigned to slots on idle worker nodes by the master while considering data locality. Similarly, the master schedules reduce tasks on idle worker nodes that read the intermediate output from the map tasks. Between the map and the reduce phases of the execution the intermediate map data are shuffled across the reduce nodes and a distributed sort

is performed. This ensures that all data with a given key are guaranteed to be redirected to the same reduce node, and in the reduce processing phase all keys are streamed in a sorted order. Re-execution of a failed task is supported where the master reschedules the task. To address the issue of a small number of tasks executing substantially slower than average and slowing down the overall job completion time, duplicate backup tasks are speculatively executed and the task that completes first is used whereas others are discarded.

2.1 Scheduling

In Hadoop all scheduling and allocation decisions are made on a task and node slot level for both the map and reduce phases. I.e., not all tasks of a job may be scheduled at once. The reason for not scheduling on a resource (node) level but on a slot level, is to allow different nodes of different capacity to offer varying numbers of slots and to increase the benefits of statistical multiplexing. The assumption is that even very complex jobs can be broken down into primitive tasks that may run in parallel on a commodity compute unit. The schedulers assume that each task in the same job takes roughly the same amount of time to complete given a slot. If this is not the case some heuristics may be applied like speculative scheduling.

All tasks are by default scheduled using a FIFO queue. Experience from large deployments at Yahoo! shows that this leads to inefficient allocations and the need for "social scheduling". The next generation scheduler in Hadoop, Hadoop on Demand (HOD), addressed this issue by setting up private MapReduce clusters on demand, managed by the Torque batch scheduling system. This approach failed in practice because it violated the data locality design of the original MapReduce scheduler, and it became too high of a maintenance burden to support and configure an additional scheduling system[1]. Creating small sub-clusters for processing individual users' tasks, as in the HOD case, violates locality because the processing nodes only cover a subset of the data nodes, and thus more data transfers are needed to stage in and out data to and from the compute nodes.

To address some of these shortcomings, Hadoop recently added a scheduling plug-in framework with two additional schedulers that extend rather than replace the original FIFO scheduler. The additional schedulers implement alternative fair-share capacity algorithms where separate queues are maintained for separate pools (groups) of users, and each are given some service guarantee over time. The inter-queue priorities are set manually by the MapReduce cluster administrator. This reduces the need for social scheduling of individual jobs but there is still a manual or social process needed to determine the initial fair distribution of priorities across pools, and once this has been set all users and groups are limited by the task importance implied by the priority of their pool. There is no way for users to optimize the usage of their granted allocation across jobs of different importance, during different job stages, or to respond to run-time anomalies such

[1] https://cwiki.apache.org/jira/browse/HADOOP-3421

as failures or slow nodes. The potential allocation inefficiency arising from this static setup is the main target for our work.

Previously we studied scheduling of entire virtual-machine-hosted Hadoop clusters in [7]. The general problem addressed there was how to scale up and down a set of virtual machines running Hadoop workers to complete jobs more cost-effectively and faster, based on knowledge of job workflow resource requirements. This approach works well if each user works with a separate data set. However, in case of groups of people sharing large data sets, it becomes too much of an overhead to load the data into multiple virtual clusters, and if file system clusters are shared you face the same problem as with HOD of reduced data locality. Furthermore, Hadoop is very IO intensive both for file system access and Map/Reduce scheduling, so virtualization incurs a high overhead. To address these problems we, in this work, focus on the approach of allocating slots in the Hadoop scheduler for different queues dynamically. This approach works both in a virtual and physical cluster, and it incurs less overhead when sharing the cluster among a large number of users. Next we describe our scheduler design and implementation in more detail.

3 Design

The primary design goal of our Hadoop task scheduler is to allow capacity distribution across concurrent users to change dynamically based on user preferences. Traditional priority systems that try to guess user priority are too inaccurate [8], and unregulated user priorities assume trusted small groups of users. Our scheduler automates capacity allocation and redistribution in a regulated task slot resource market.

3.1 Mechanism

The core of our design is a proportional share resource allocation mechanism that allows users to purchase or be granted a *queue priority budget*. This budget may be used to set *spending rates* denoting the willingness to pay a certain amount of the budget per Hadoop map or reduce task slot per time unit. The time unit is configurable, and referred to as *allocation interval*. It is typically set to somewhere between 10 seconds and 1 minute. In each allocation interval the scheduler:

- aggregates all spending rates s from all current users to calculate the Hadoop cluster *price*, p,
- for all users, allocates $(s_i/p) \times c$ task slots (both mappers and reducers) to user i, where s_i, is the spending rate of user i, and c is the aggregate slot capacity of the cluster,
- for all users, deducts $s_i \times u_i$ from budget b where u_i, is the number of slots used by user i

Users consuming more resources will deplete their budget faster given the same spending rate. However, they are guaranteed to not pay more than the spending

rate per allocated slot. Thus a user's *bid* represents her willingness to pay a certain rate per slot.

It may appear that this model is biased towards users with small jobs who would be able to outbid users with bigger jobs. However, in the Hadoop MapReduce task model users with big jobs can effortlessly scale down their jobs to run fewer concurrent tasks and thereby consume the same amount of resources per time unit as small jobs but instead run longer. Our model thus sets the right incentives for users to scale back resource consumption as much as their job deadlines or SLAs allow.

Because we only want to charge each user for the capacity they use and reallocate the unused capacity to other users, (and we want to make sure users actually pay for the spending rate they *bid*) we calculate the capacity allocation and the price to pay for slots for an allocation interval based on the spending rates in the interval directly preceding the interval when the slots are consumed. To avoid blocking new arriving users and having non-running users hold up resources, we only calculate an allocation for a user if either a job is pending or running for that user.

To adapt more quickly to user demand fluctuations and avoid head of queue blocking and starvation issues, we support preemption where task slots that have been allocated but are no longer paid for may be reclaimed and allocated to other users. This works well for most applications since Hadoop automatically puts preempted tasks back in the pending queue to be reallocated when demand, measured by user spending rates, allows.

The key feature of this mechanism is that it discourages free-riding and gaming by users. Users who claim a higher priority will have to pay for it, so they have an incentive to accurately reveal how important priority is to them. In addition, the variable pricing allows users with a low budget and low time-sensitivity to run during low demand periods. These users would otherwise not be able to run at all in a fixed pricing model. Conversely, at high demand periods, users have a disincentive to run, but resources will nonetheless be available (for a high price) for users that really need them.

The disadvantage is less capacity predictability and more variation in capacity allocated to an application. However, the Hadoop MapReduce scheduling framework allows jobs to be split up in finer grained tasks that can run and possibly fail and recover independently. So the only thing the end users would need to worry about is to get a good enough average capacity over some time to meet their deadlines.

This introduces the difficulty of making spending rate decisions to meet the SLA and deadline requirements. It is outside the scope of this paper and the target of future work to address this particular issue, but the mechanisms presented here opens the door for innovation in this area, by allowing much more fine grained control over resources for competing users in a multi-tenancy hosted Hadoop cluster.

Figure 1 depicts how our scheduler components fit into the Hadoop architecture. Alice is willing to pay $4 per slot, Bob is willing to pay $1.50, and Sam $2.

Assuming that 15 slots are available to these three users in the global (logical) slot table, Alice will be allocated 8 slots, Bob 3 slots and Sam 4 slots. Exactly how these slots are mapped to physical nodes is not guaranteed. Whenever a slot becomes available the allocations are recalculated to determine who should get the new slot according to their granted share. Furthermore, local tasks are attempted first. If that fails, remote rack tasks are scheduled. There may be opportunities to delay scheduling of some jobs to achieve a higher ratio of data local tasks. However, in the current implementation we enforce the shares strictly in each time period. This is not overly restricting because Hadoop replicates all the data in at least three data blocks by default, which ensure many opportunities for data local scheduling. Packing a user on a single node versus distributing the job workload across nodes is another application specific trade-off that we may address in future implementations.

Possible starvation of low-priority (low-spending) tasks can be mitigated by using the standard approach in Hadoop of limiting the time each task is allowed to run on a node. Moreover, our new mechanism also allows administrators to set budgets for different users and let them individually decide whether the current price of preempting running tasks is within their budget or if they should wait until the current users run out of their budget. The fact that Hadoop uses task and slot level scheduling and allocation as opposed to job level scheduling also avoids many starvation scenarios.

If there is no contention, i.e. there are enough slots available to run all tasks from all jobs submitted, the cost for excess resources essentially becomes free because of the work conserving principle of our scheduler. However, the

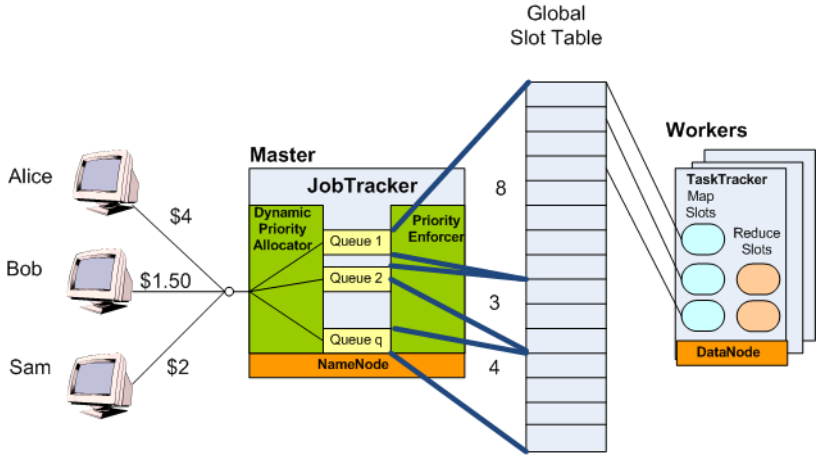


Fig. 1. Dynamic Priority Scheduler Architecture. This example shows how a max capacity of 15 Map slots gets allocated proportionally to three users. For example, Alice bids \$4 and gets $4/(4 + 1.5 + 2) * 15 = 8$ slots. The central scheduler comprises a Dynamic Priority Allocator and a Priority Enforcer component responsible for accounting and schedule enforcement respectively.

guarantees of maintaining these excess resources are reduced. To see why, consider new users deciding whether to submit jobs or not. If they see that the price is high they may wait to preempt currently running jobs, but if the resources are essentially given out for free they are likely to lay claim on as many resources they can immediately.

We note that the Dynamic Priority scheduler can easily be configured to mimic the behavior of the other schedulers. If no queues or users have any credits left the scheduler reduces to a FIFO scheduler. If all queues are configured with the same share (spending rate in our case) and the allocation interval is set to a very large value, the scheduler reduces to the behavior of the static fair-share schedulers.

3.2 Implementation

The *Dynamic Priority* scheduler is implemented as a scheduler plugin for the Hadoop JobTracker service. This allows DP to be a drop-in replacement of the default FIFO scheduler. The scheduler is split into two components: one for allocation, *Dynamic Priority Allocator*, and one for enforcement, *Priority Enforcer*.

The *Dynamic Priority Allocator* implements dynamic slot allocation, budgeting and accounting, and provides a remote secure API to manage and monitor budgets and spending rates.

The *Priority Enforcer* component is responsible for enforcing the shares of resources calculated by the allocation component. It is responsible for picking pending tasks from jobs to be scheduled when mapper and reducer slots open up in Hadoop TaskTrackers. It thus implements the same functionality as the FIFO and fair-share schedulers. However, these schedulers were not designed to handle a large number of queues with constantly varying capacities that are determined on demand from user input. They do not enforce shares at the granularity and precision that our mechanism requires and do not support preemption to the extent that we require.

The budgets and spending rates are stored in a storage component that can be file-based or SQL-based. An XML REST Servlet controls the scheduler. The monitoring component plugs into the Hadoop JobTracker Web console. The Web console is depicted in Figure 2. The numbers displayed next to each queue

Table 1. REST XML API to Manage Scheduler Allocations

HTTP Options	**Description**	**Authz**
price	Gets current price	None
info=*queue*	Gets queue usage info	User
infos	Gets usage info for all queues	Admin
setSpending=*spending*&queue=*queue*	Set the spending rate for queue	User
addBudget=*budget*&queue=*queue*	Add budget to queue	Admin
addQueue=*queue*	Add queue	Admin
removeQueue=*queue*	Remove queue	Admin

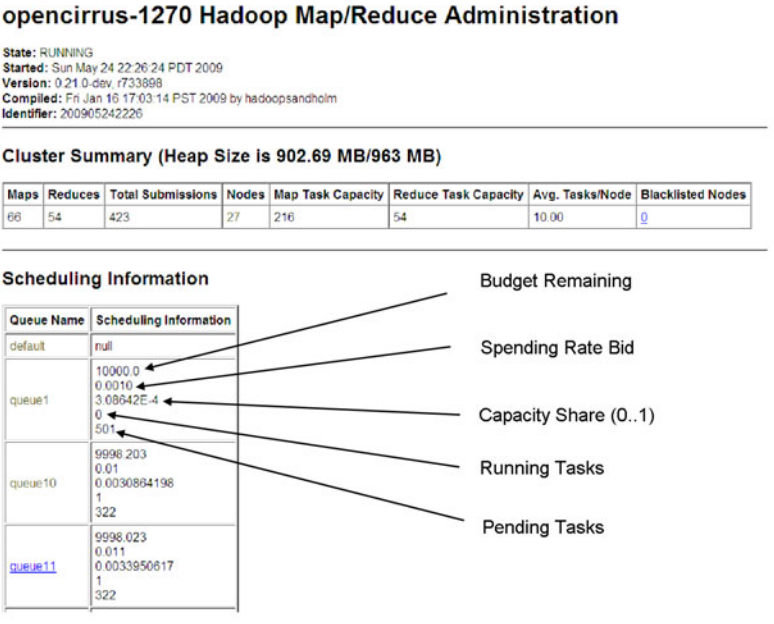


Fig. 2. MapReduce Administration Monitor

represent from top to bottom: current budget, spending rate, resource share, slots used, and slots pending. The supported APIs are listed in Table 1 and an example XML response for authorized requests can be seen in Listing 1.

Listing 1. Example XML response for authorized requests

```
<QueueInfo>
  <host>myhost</host>
  <queue name="queue1">
    <budget>99972.0</budget>
    <spending>0.11</spending>
    <share>0.008979593</share>
    <used>1</used>
    <pending>43</pending>
  </queue>
</QueueInfo>
```

3.3 Security and Authentication

The existing Unix user and group based security model of Hadoop is too simple to support a full-fledged multi-tenancy resource market as described above. More specifically, relying on each user to pick queues and be trustworthy about their identity would defeat the accounting and budget enforcement mechanism. As a result, we implemented a lightweight symmetric key authentication and role-based authorization protocol modeled after AWS Query Authentication [9], and

OAuth. The advantage is that it is easy to use from any client and only requires the capability to construct HMAC/SHA1 signatures based on shared secret keys. The existing Hadoop command line clients were also extended to pass the signatures required to submit jobs to queues being paid for in job configuration parameters.

4 Evaluation

In this section, we describe experiments run to study the scalability and allocation dynamics of our scheduler. There are three sets of experiments. In the first set, we examine the correlation of spending rates, budgets and performance metrics. In the second set, we study how accurately and effectively service levels can be supported. Finally we measure how well the system adapts to changes in spending rates. Unless otherwise stated all users are given the same budgets in all experiments. We use the term *queue* interchangeably with the term *user* since all users are given a dedicated queue to submit their jobs on in all of our experiments. Our scheduler allows queues to be shared across users but it should be compared to sharing bank accounts or access to a PC account among users, i.e. sharing security credentials such as passwords, which is generally frowned upon.

4.1 Setup

We use two testbeds for our evaluation: a 30 node quad-core cluster (referred to as the *big* cluster) and a 5 node octo-core cluster (referred to as the *small* cluster). The *small* cluster runs on virtual machines, whereas the *big* cluster is installed directly on the hardware. More details of the clusters are shown in Table 2.

For both setups, we allocate one queue per user and run 2-80 users concurrently. All users run the same benchmark application, the Pi estimator from the Hadoop example code base. The Pi application was set up to be able to consume the entire cluster if run in isolation (i.e. number of job tasks were set to the number of slots available in the cluster), and thus ran slower when there was contention. The pi precision target was set to 450000000 for the small cluster and 500000000 for the big cluster to ensure that the application was both CPU and data intensive. The ability to fine-tune the CPU versus data intensity without having to provision a large amount of data was the main reason we chose the Pi application for our experiments. The fact that all Hadoop applications conform to the same general internal structure (MapReduce) allows us to treat the results more generally than with a typical parallel workload. To stress the system, all users are launched concurrently and submit a continuous stream of jobs. In the initial 2-user experiments we test the FIFO, Fairshare (fair-share scheduler developed at Facebook), and Capacity (fair-share scheduler developed at Yahoo!) schedulers and compare them to the Dynamic Priority scheduler that we developed. The Fairshare and Capacity schedulers were not able to handle the 10-80

Table 2. Experiment Cluster Setup

Cluster	Used in Graphs	Nodes	Cores (CPUs)	Physical/Virtual	OS	Disk
big	3-9	30	120(30)	Physical	CentOS 5	45TB
small	10-11	5	40(40)	Virtual	CentOS 5	250GB

queue and user workload reliably so they were excluded from the larger experiments. To switch between the schedulers during the experiment we restarted the JobTracker service resulting in a clean start since no running job information is persisted in the current version of the JobTracker. The stream of jobs from the clients is not affected either during a restart since the clients will just resubmit jobs when a job is done or fails.

4.2 Spending Rates, Budgets and Performance

In the first experiment, we start two concurrent user workloads. We give queue1 an initial budget of 1000 and queue2 10000 credits. The spending rate per Hadoop slot of queue1 is set to twice the rate of queue2. Since queue1 will then be allocated twice as many resources the total spending is expected to be 4 times that of queue2 in any allocation interval.

Figure 3 depicts the budget over time for the two users, and Figure 4 shows the completion time of their jobs over the same time period. Our scheduler is initially configured to run without preemption and queue1 will thus not see an immediate benefit in completion time.

We also see that the budget of queue1 runs out at time 05/15-22:00, at which point the allocation is given over to queue2, and the performance of queue1 degrades significantly. At time 05/16-14:00 the budgets of queue1 and queue2 are reset to 10000 and the scheduler is reconfigured to preempt. We now see that the queue1 completion time is around 3000s for each job in Figure 4 and the spending is about 26-27% more than queue2 (25% expected) as seen in Figure 3. We do not obtain exactly half the job completion time when getting twice the amount of resources but about 1.8. This is because we only control the slot capacity not other resources such as HDFS (distributed file system) IO and network bandwidth. We can also see that the higher spender (queue1) gets a very stable high performance, oscillating between 3000-3200s completion times compared to the low priority queue (queue2) which oscillates between 4500-5800s.

Now just looking at Figure 4 at time 05/18-00:00 we reconfigure the cluster to use the Capacity scheduler. The differentiation in obtained service level is far less although the capacity configuration is the same, twice as many slots for queue2. We attribute this to less aggressive preemption, and less granular control over allocations in this scheduler compared to ours. We can also see that the min/max range variation is greater for both queues with the capacity scheduler. Queue1 oscillates between 3000-3600s, and queue2 oscillates between 3600-5500s.

Taking the ratio of minimum performance to maximum performance we get a differentiation of about 1.5 to be compared to 1.8 for our scheduler. At time

05/19-00:00 we had a failed attempt to set up the Fairshare scheduler for this workload. We saw that all schedulers showed signs of a memory bloat with workload and would eventually run out of memory. This behavior was most apparent with the Fairshare scheduler which did not manage to complete a single job. We point out that this bug was not in any of the schedulers but in the jobtracker framework, so it just surfaces how different schedulers handle memory in general. So instead at time 05/19-18:00 we reconfigure the cluster with the standard FIFO scheduler. We can see that this scheduler does not offer any differentiation as expected, and the average performance level is above the queue1 level and below the queue2 level obtained with the other schedulers.

We note that the capacity scheduler was configured with 60min preemption. More frequent preemption caused problems with completing the tasks. Neither the fair-share nor the FIFO schedulers supported preemption in the versions tested[2]. However, both Capacity and Fairshare Queue/Pool capacity was configured the exact same way as with our scheduler, with the only difference that it was not able to change over time. The FIFO scheduler was not configured with any priorities, since no queue-based priorities could be set.

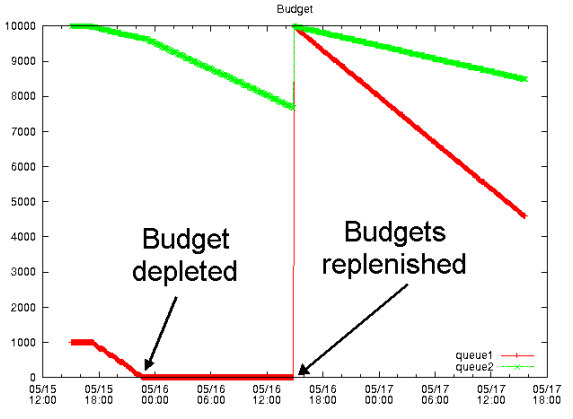


Fig. 3. *2-user budget dynamics example.* The graph shows how the budget (y-axis) evolves over time (x-axis as month/day and time). The slopes of the curves represent the spending rates of the users over time. Queue(user)1 uses twice the spending rate of queue(user)2. At the center of the graph the budgets of both users are reset to 10000 (time 05/16 14:00).

We stress that it is not simply an implementation artifact that the capacity and fair-share schedulers perform poorly in these tests. These schedulers were not designed for dynamic priorities nor for handling a large number of queues from the outset as our scheduler was[3].

[2] Hadoop 0.20-0.21 code base checked out around May 2009

[3] http://issues.apache.org/jira/browse/HADOOP-4768

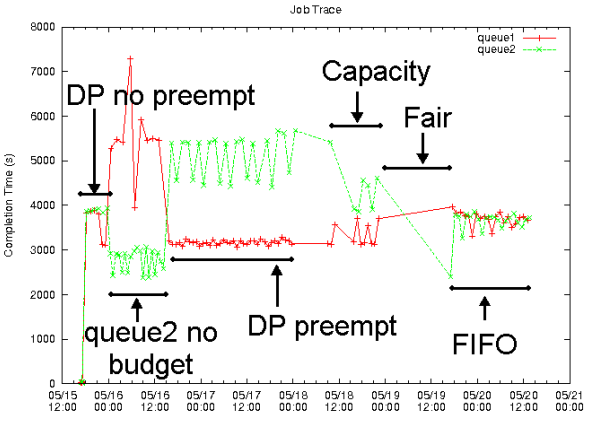


Fig. 4. *2-users service differentiation trace.* The graph shows the completion time over time for jobs submitted by the 2 users in the budget graph in Figure 3. The first half of the timeline corresponds directly to the timeline in the budget graph. The second half corresponds to experiments with the capacity, fairshare and FIFO schedulers. The first drop in completion time for queue1 is correlated with the budget running out. The key result is the clear separation of completion times between queue1 and queue2 seen in the first half compared to the second half of the graph.

4.3 Allocation Fidelity and Overhead

Now we look at how well we can preserve the differentiation of service levels with more users and queues. Figure 5 shows the completion times obtained for 10 queues when queue n is given a share of $n/\sum_{i=1}^{10} i$. We can see that all 10 service levels are enforced successfully. At time 05/21-10:00 we reconfigure the cluster with the FIFO scheduler. We note that there is a random distribution of service levels for the first job because there is no preemption. For other jobs the identical service level is given to all jobs. This experiment showcases that a dynamic non-stationary workload with users entering and leaving the system may result in random highly variable service levels even with the FIFO scheduler.

In Figure 6 we show the results of an experiment that ran our scheduler with preemption and 80 users first, then the FIFO scheduler and finally our scheduler without preemption. Still we see that the 10 service levels are maintained. We do not obtain more than 10 service levels with this application (Pi estimator). The number of service levels obtainable depends both on overhead and bottlenecks in the specific applications run but also on the overall scale of the cluster and the slots available. We also note here that the preempting version of our scheduler, in the left half of the graph, delivers somewhat more stable service level than the non-preemptive one (after time 05/24-22:00) but the differences are cosmetic. This experiment again shows that our scheduler shapes the workflow into the desired service levels quickly.

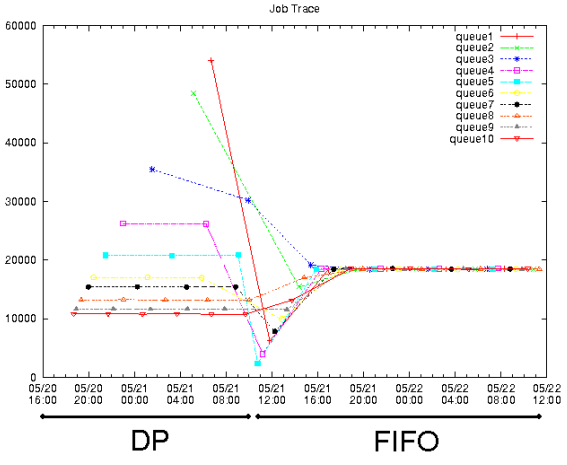


Fig. 5. *10-user service differentiation trace.* The graph shows completion time for jobs (y-axis) over time (x-axis). The first half of the graph shows how our scheduler separates the queues' performance compared to the second half when the FIFO scheduler was used. Half of the queues obtain better performance and the other half worse than the FIFO case.

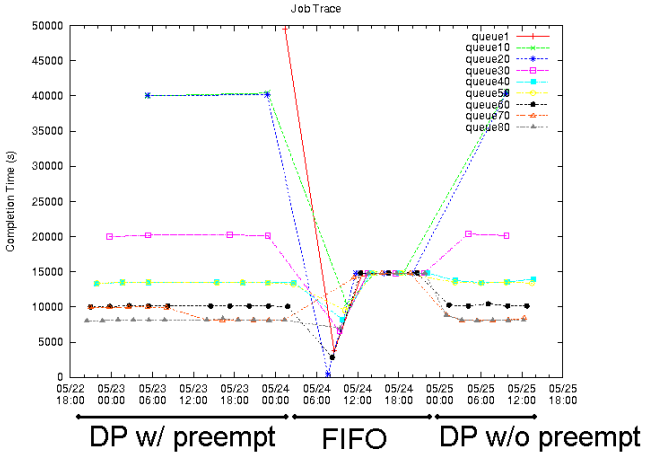


Fig. 6. *Sample of 80-user service differentiation trace.* The graph shows completion time (x-axis) over time (y-axis) using the same setup as in the 2-user graph in Figure 5, but with 80 users. For clarity only a sample of the users are shown. The results are very similar to the 2-user graph, which shows how our scheduler's ability to differentiate service levels scales well in number of queues/users.

Table 3. Distance to Ideal Line (in seconds) from Average Queue Completion Time with Approximate 95% Confidence Bounds

Scheduler	Queue1	Queue2
Capacity	1000 ± 150	600 ± 250
DynPrio	800 ± 20	300 ± 200

We now study the performance fidelity of the granted allocation more carefully. There is obviously some trade-offs in throughput of the system and the level of preemption enforced since a killed Hadoop task (note not a job) must be restarted from the beginning. Figure 7 shows the fidelity versus overhead for the two-user experiment. The ideal line depicts the performance expected if queue1 runs its jobs twice as fast as queue2, but the average across the queues is the same as for the FIFO case (e.g. optimal fidelity and maximum throughput). Our dynamic priority scheduler running with preemption comes closest to meeting this ideal, but we can also see that we can improve the throughput and move closer to the FIFO line if preemption is not turned on. Improved closeness to ideal here is seen by observing that both the queue1 point and the queue2 point in the graph for the 60s preempt dynprio line are closer to the respective ideal line points (see also Table 3).

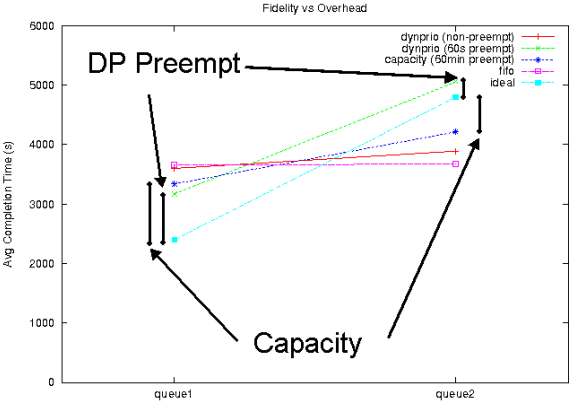


Fig. 7. *2-user fidelity to granted shares and throughput loss.* This graph compares the overhead of differentiating service levels to using FIFO scheduling. The fairshare scheduler was not included in the results due to reliability issues. However, it behaves similarly to the capacity share scheduler. The ideal line represents the performance that should have been observed for the two queues if adhering to the configured capacities while obtaining the same throughput as with the FIFO scheduler. When comparing the slopes of the dynprio preempt line and the capacity scheduler line with the ideal line we see that the slope of the dynprio line is a closer match (one of the goals of our scheduler).

One could argue that the capacity scheduler achieves the least degradation across both users while still achieving some differentiation and should therefore be preferred. This may be the case in fair-share scheduled systems where users do not pay for their usage. But in a cloud computing scenario where queue1 actually paid twice as much as queue2 it may no longer hold true. We focus more on differentiating service-levels that are as close as possible to the capacity you pay for as opposed to achieving some overall fair outcome in our scheduler.

Figure 8 shows the corresponding graph for the 10 user experiment. We can see that the extremes (highest and lowest service levels) are far away from the ideal line whereas service levels 3 through 9 mimic the ideal scenario well. We also show an ideal adjusted line that has the same service level as the dynamic priority scheduler for the maximum service level but the same degradation in service levels as the ideal line. We can see that only service levels 1 and 2 fall outside of the ideal and ideal adjusted lines, which indicates that our scheduler is a bit biased against users with low spending rates. The same behavior can be seen in the 80-user experiment depicted in Figure 9. Here we note that the users are heavily discretized in groups of about 10-15. This is most likely due to the MapReduce workload chosen which only uses 10 reducers, and thus limits the reduce phase throughput to 10 service levels.

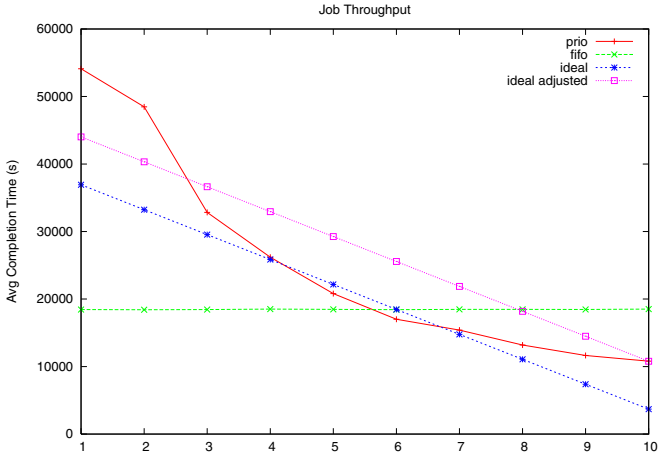


Fig. 8. *10-user fidelity to granted shares and throughput loss.* This graph compares the overhead of differentiating service levels to using FIFO scheduling for the experiment with 10 users (user 1-10 denoted on x-axis). The ideal adjusted line corresponds to the ideal (no overhead and perfect differentiation) line with the same minimal completion time as observed in the experiments. Only users 1 and 2 (with the lowest slot capacity) deviate significantly from the ideal lines.

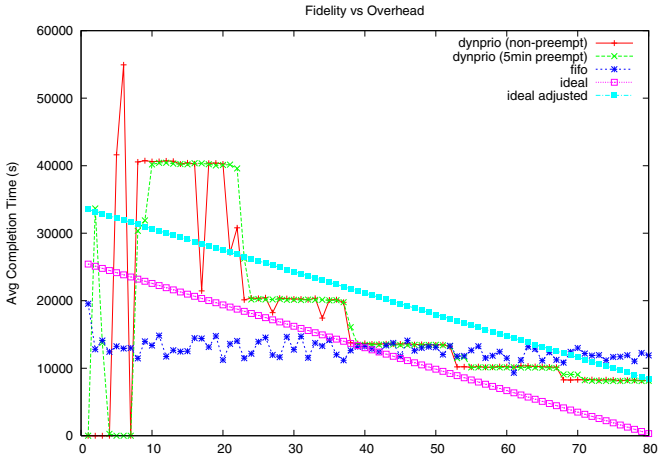


Fig. 9. *80-user fidelity to granted shares and throughput loss.* This graph compares the overhead of differentiating service levels to using FIFO scheduling for the experiment with 80 users (user 1-80 denoted on x-axis). As in the 10-user graph the top 80 percent of the users (with highest spending rates and capacity) obtain completion times within the ideal lines (right side of graph).

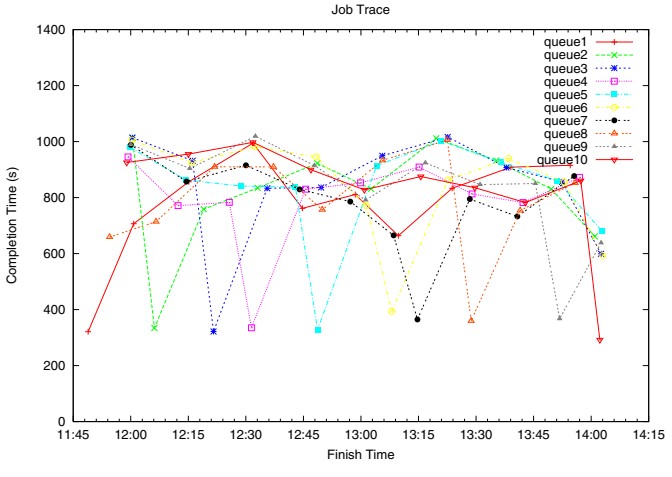


Fig. 10. *Dynamic priority adjustment with 10 users, with 60s preemption.* The graph shows the completion times of jobs for queues/users who increased their spending rates for their x'th job, where x is the queue number. All boosted jobs obtained a significant decrease in completion time, showing the agility and dynamic nature of our scheduler.

4.4 Adaptability of Service Levels

We run the final experiments on our *small* cluster and investigate how well we can dynamically adjust the service levels. 10 users all run 10 Pi estimator jobs in sequence and concurrent with all the other users. User n is given a 4x boost

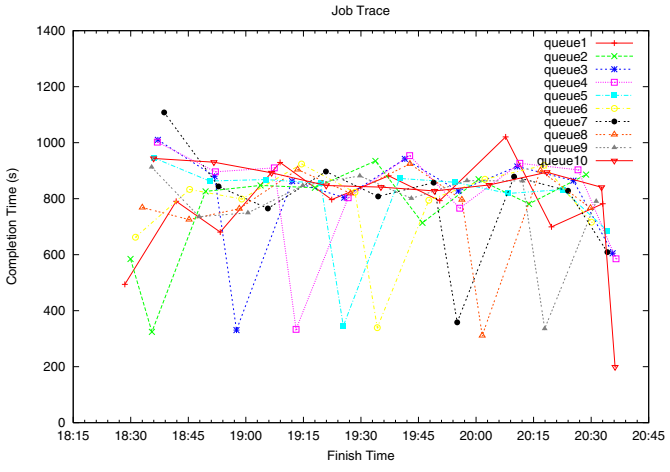


Fig. 11. *Dynamic priority adjustment with 10 users, without preemption.* The graph shows the completion times of jobs for queues/users who increased their spending rates for their x'th job, where x is the queue number when no preemption was used. The graph looks almost identical to the preemption graph with only slight deviations for the boosted jobs from user 1 and user 10.

in spending rate for job n. In Figure 10 we can see that a 3x performance boost is obtained consistently for all users and jobs regardless of when during the job sequence the boost kicks in. The valleys hover around completion times of 300s, whereas the average of non-valley jobs lies around 900s. Figure 11 shows the same experiment but with preemption turned off. We can then see that the service levels of the first jobs are random but all other jobs follow the same pattern as in the preemption case. This shows that we are able to converge quickly to a stable state even without preemption. The overhead of preemption, calculated based on the difference in average job completion time between the two experiments was less than 2.6%.

5 Discussion

Some issues merit additional discussion: preemption, dynamic adjustment, and currency management.

Whether preemption should be offered or not depends on the types of workloads expected. For CPU bound, embarrassingly parallel applications that benefit from holding a slot for a longer duration of time, preemption may be necessary to avoid starvation effects. On the contrary, for data bound applications that stream small amounts of input at a time into Map and Reduce tasks that complete within a couple of minutes, preemption may not add much value. We saw that using preemption incurred a small (2.6%) overhead in throughput, but allowed the system to adhere to service levels more quickly and accurately.

Note that preemption in the Hadoop context is somewhat different from the traditional CPU/scheduling type of preemption. Hadoop preemptions do not suspend and then resume the task but rather kills the task and forces it to start over again. It thus causes a throughput penalty. Care must hence be taken to kill the jobs that will degrade the throughput the least while still ensuring that starvation and unfairness effects are minimized.

One feature of the Dynamic Priority scheduler (DP) is that it allows users to change the priority of jobs during a run. However, it does not require it. Users who prefer not to monitor their jobs can let them run as initially configured. The opportunity to change priorities is most useful to handle unexpected situations like server failures, increases in load by other users, and the inability of users to predict their own job runtimes. In the latter case, DP allows users to adjust their spending rates so that the actual running time of their jobs fits their deadlines.

Since the DP introduces a currency into the system, it requires the system administrator to manage the overall economy of the system. The basic goal is to keep a stable exchange rate between currency and computational work. Users need to be able to expect that 1 credit will generally get 1 server hour (for example). Of course, prices will fluctuate, but the average should remain stable. Admins can do this by setting a total income rate per hour for the system which is equal to the number of servers. The admin then distributes this income among the users. For example, a cluster of 200 servers would have an income of 4800 credits per day which can be allocated for users. This total is fixed, regardless of the number of users, so the admin should reserve some amount for new users. As the admin adds new servers, the total can increase.

If prices start increasing significantly, this indicates that the system is under-provisioned with respect to its load. The admin should consider adding more servers and/or moving some users to another system. Conversely, if prices collapse, then the system is over-provisioned and the admin can add users and/or remove servers.

The admin must be careful with the inevitable demands to increase the income rate for some users. If some users actually have more important jobs than other users, then the admin should increase the income rate of the important users while decreasing the rate for other users such that the total income rate is the same. Otherwise, the system will enter an inflationary spiral that is difficult to break out of.

6 Related Work

Parallel job scheduling is a well-investigated field both in theory and in practice with applications beyond computational resource management [10]. Theoretical studies commonly assume embarrassingly-parallel jobs which has lead to much of the innovation in the field to be driven by simulations and experiments [11]. The most commonly deployed scheduling regime is First-Come-First-Served (FCFS) or variations thereof. FCFS suffers from head of queue blocking and starvation issues. Two popular variations to address these issues are backfilling [12] and

gang scheduling [13] [14]. Many heuristics and variations have been proposed to improve throughput, e.g. Shortest-Job-First (SJF), or fairness, e.g. Fair-Share Scheduling. Many of these classical scheduling algorithms focus on improving systems metrics such as utilization and average response time. Some of these systems may however be very inefficient in terms of serving the most important task at the best time from an end-user point of view. The reason for this is that priorities are either assigned by the system, or are only valid across jobs for the same user, as exemplified by the Maui scheduler [15].

Proportional share and Lottery scheduling were proposed in [8] to give users more direct and dynamic control over capacity allocations for different types of tasks over time. In previous work this technique has been applied to both cluster node [16] and VM resource scheduling [17]. To our knowledge our work is the first applying the proportional share mechanism to MapReduce slot scheduling for computational clusters.

Our scheduling approach is closely related to and inspired by economic schedulers, whereby you bid for resources on a market and receive allocations based on various auction mechanisms [18,19,20,17,21,22,23,24]. We do not preclude nor require that our scheduler budgets are tied to a real currency. Furthermore, we do not assume that there are competing users who should be given different shares of the resources. Giving all users the same budget initially but allowing them to spend this budget at different rates is a valid use case of our scheduler. Many game theory inspired agent scheduling algorithms such as the Best Response algorithm in [25], could be implemented on top of our scheduler for Hadoop jobs. Meta-scheduling across Hadoop clusters in different organization is also simplified by exposing different demand-based prices for running jobs in a cluster.

Other work to improve the FIFO and fair share scheduling in Hadoop includes the LATE scheduler [26]. The main purpose of the LATE scheduler is to predict Hadoop job progress more accurately and to take overhead into account when launching speculative tasks. In [27] the work on the LATE scheduler is extended by two new techniques, delay scheduling and copy-compute splitting, designed to improve data locality and avoid reduce slot bottlenecks respectively. These techniques are complimentary to our work. In theory both of these issues are orthogonal to our scheduling mechanism since they tackle separate problems (not incentives and accountability which are at the core of our work). In practise, the delayed scheduling technique would require some changes in how slots are allocated in our scheduler, but since we only charge for slots that are actually used, the general accounting mechanism would stay the same.

MapReduce scheduling has also been explored beyond the traditional data center domain, such as for Cell [28], GPUs [29], and shared memory architectures [30]. Our general proportional share MapReduce slot algorithm presented in this paper could thus potentially also be employed in these other domains.

7 Conclusion

Our experimental results demonstrate that our scheduler scales better than the existing Hadoop schedulers in the number of queues. Having more queues

allows providers to provide more service levels. The fair-share scheduler could not even handle the experimental workload for two concurrent queues, whereas the capacity scheduler was not able to handle the workload with ten queues. The Dynamic Priority scheduler handles up to 80 queues efficiently, which was only limited by the memory capacity of the experiment client node.

This enhanced scalability is due to the light-weight design of DP. In contrast to the other schedulers, it does not incur the overhead of heuristics for inferring fair priorities over time. Instead, DP users directly decide priorities, so it only has to maintain the budget currently remaining. As of this writing, the capacity scheduler contains 140KB of non-test source code, the fair-share scheduler 130KB, and DP 55KB.

Furthermore, we have shown that DP adapts service levels dynamically and quickly even during heavy load, adheres to them more accurately. This was shown by having 10 users with a stream of 10 15min jobs all boost their single high priority jobs accurately without overhead or notable randomness.

DP also solves the problems of lost data locality and virtualization overhead that we encountered in our previous work on virtualized MapReduce [7]. The downside is that we lose some control over tasks that are long-running, and the isolation properties cannot be enforced as strictly. However, an advantage is that it becomes easier to provision commonly used software and data sets in shared test-beds.

Future work includes leveraging the dynamic capacity control in our scheduler to adaptively change the allocations to meet higher level SLA goals such as deadlines.

References

1. White, T.: Hadoop: The Definitive Guide. O'Reilly, Sebastopol (2009)
2. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: Symposium on Operating System Design and Implementation (2004)
3. Ghemawat, S., Gobioff, H., Leung, S.-T.: The Google File System. In: ACM Symposium on Operating Systems Principles (2003)
4. Bryant, R.E.: Data-intensive supercomputing: The case for DISC. Technical Report CMU-CS-07-128, Carnegie Mellon University (2007)
5. http://wiki.apache.org/hadoop/PoweredBy (2009)
6. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A Distributed Storage System for Structured Data. In: Symposium on Operating System Design and Implementation (2006)
7. Sandholm, T., Lai, K.: Mapreduce optimization using regulated dynamic prioritization. In: SIGMETRICS 2009: Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, pp. 299–310. ACM, New York (2009)
8. Waldspurger, C.A.: Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management. Technical Report MIT/LCS/TR-667 (1995)
9. Amazon elastic compute cloud (2008), http://aws.amazon.com/ec2 (retrieved March 6, 2008)

10. Pinedo, M.: Scheduling: Theory, Algorithms, and Systems, 3rd edn. Springer Science, Heidelberg (2008)
11. Frachtenberg, E., Schwiegelsohn, U.: New Challenges of Parallel Job Scheduling. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2007. LNCS, vol. 4942, pp. 1–23. Springer, Heidelberg (2008)
12. Lifka, D.: The ANL/IBM SP scheduling system. In: Feitelson, D., Rudolph, L. (eds.) IPPS-WS 1995 and JSSPP 1995. LNCS, vol. 949, pp. 295–303. Springer, Heidelberg (1995)
13. Ousterhout, J.K.: Scheduling techniques for concurrent systems. In: 3rd International Conference on Distributed Computing Systems, pp. 22–30 (1982)
14. Feitelson, D.G., Rudolph, L., Schwiegelshohn, U.: Parallel job scheduling - a status report. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2004. LNCS, vol. 3277, pp. 1–16. Springer, Heidelberg (2005)
15. Jackson, D., Snell, Q., Clement, M.: Core algorithms of the maui scheduler. In: 7th International Workshop on Job Scheduling Strategies for Parallel Processing, pp. 87–102 (2001)
16. Chun, B.N., Culler, D.E.: Market-based proportional resource sharing for clusters. Technical Report CSD-1092, University of California at Berkeley, Computer Science Division (2000)
17. Lai, K., Rasmusson, L., Adar, E., Sorkin, S., Zhang, L., Huberman, B.A.: Tycoon: an implemention of a distributed market-based resource allocation system. Multiagent and Grid Systems 1, 169–182 (2005)
18. Ernemann, C., Yahyapour, R.: Applying economic scheduling methods to grid environments. In: Grid Resource Management: State of the Art and Future Trends, pp. 491–506 (2004)
19. Piro, R.M., Guarise, A., Werbrouck, A.: An economy-based accounting infrastructure for the datagrid. In: GRID 2003: Proceedings of the 4th International Workshop on Grid Computing, Washington, DC, USA, p. 202. IEEE Computer Society, Los Alamitos (2003)
20. Waldspurger, C.A., Hogg, T., Huberman, B.A., Kephart, J.O., Stornetta, W.S.: Spawn: A Distributed Computational Economy. Software Engineering 18, 103–117 (1992)
21. Chun, B.N., Culler, D.E.: User-centric performance analysis of market-based cluster batch schedulers. In: Proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid (2002)
22. Sandholm, T., Lai, K., Clearwater, S.: Admission control in a computational market. In: CCGrid 2008: Proceedings of the 8th International Symposium on Cluster Computing and the Grid (2008)
23. Wolski, R., Plank, J.S., Bryan, T., Brevik, J.: G-commerce: Market formulations controlling resource allocation on the computational grid. In: IPDPS 2001: Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS 2001), Washington, DC, USA, p. 10046.2. IEEE Computer Society, Los Alamitos (2001)
24. Buyya, R., Murshed, M., Abramson, D., Venugopal, S.: Scheduling Parameter Sweep Applications on Global Grids: A Deadline and Budget Constrained Cost-Time Optimisation Algorithm. Software: Practice and Experience (SPE) Journal 35, 491–512 (2005)
25. Feldman, M., Lai, K., Zhang, L.: A price-anticipating resource allocation mechanism for distributed shared clusters. In: Proceedings of the ACM Conference on Electronic Commerce (2005)

26. Zaharia, M., Konwinski, A., Joseph, A.D., Katz, R., Stoica, I.: Improving MapReduce performance in heterogeneous environments. In: OSDI 2008: 8th USENIX Symposium on Operating Systems Design and Implementation (2008)
27. Zaharia, M., Borthakur, D., Sarma, J.S., Elmeleegy, K., Shenker, S., Stoica, I.: Job scheduling for multi-user mapreduce clusters. Technical Report UCB/EECS-2009-55, Electrical Engineering and Computer Sciences University of California at Berkeley (2009)
28. Rafique, M.M., Rose, B., Butt, A.R., Nikolopoulos, D.S.: Cellmr: A framework for supporting mapreduce on asymmetric cell-based clusters. Parallel and Distributed Processing Symposium, International, 1–12 (2009)
29. He, B., Fang, W., Luo, Q., Govindaraju, N.K., Wang, T.: Mars: a MapReduce framework on graphics processors. In: PACT 2008: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, pp. 260–269. ACM, New York (2008)
30. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating MapReduce for multi-core and multiprocessor systems. In: HPCA 2007: IEEE 13th International Symposium on High Performance Computer Architecture, pp. 13–24 (2007)

The Importance of Complete Data Sets for Job Scheduling Simulations

Dalibor Klusáček and Hana Rudová

Faculty of Informatics, Masaryk University
Botatanická 68a, Brno, Czech Republic
{xklusac,hanka}@fi.muni.cz

Abstract. This paper has been inspired by the study of the complex data set from the Czech National Grid MetaCentrum. Unlike other widely used workloads from Parallel Workloads Archive or Grid Workloads Archive, this data set includes additional information concerning machine failures, job requirements and machine parameters which allows to perform more realistic simulations. We show that large differences in the performance of various scheduling algorithms appear when these additional information are used. Moreover, we studied other publicly available workloads and partially reconstructed information concerning their machine failures and job requirements using statistical and analytical models to demonstrate that similar behavior is also expectable for other workloads. We suggest that additional information about both machines and jobs should be incorporated into the workloads archives to allow proper and more realistic simulations.

Keywords: Grid, Cluster, Scheduling, MetaCentrum, Workload, Failures, Specific Job Requirements.

1 Introduction

Large computing clusters and Grids have become common and widely used platforms for the scientific and the commercial community. Efficient job scheduling in these large, dynamic and heterogeneous systems is often a very difficult task [1]. Development or application of an efficient scheduling algorithm requires a lot of testing and evaluation before such solution is applied in the production system. Due to several reasons, such as the cost of resources, reliability, varying background load or the dynamic behavior of the components, experimental evaluation cannot be usually performed on the real systems. Many simulations with various setups that simulate different real-life scenarios must be performed using the same and controllable conditions to obtain reliable results. This is hardly achievable in the production environment.

Usually, workload traces from the Parallel Workloads Archive (PWA) [2] or Grid Workloads Archive (GWA) [3] are used as the simulation inputs. However, these data do not contain several parameters that are important for realistic simulations. Typically, very limited information is available about the Grid or

E. Frachtenberg and U. Schwiegelshohn (Eds.): JSSPP 2010, LNCS 6253, pp. 132–153, 2010.

cluster resources such as the architecture, the CPU speed, the RAM size or the resource specific policies. However, these parameters often significantly influence the decisions and performance of the scheduler. Moreover, no information concerning background load, resource failures, or specific users' requests are available. In heterogeneous environments, users often specify some subset of machines or clusters that can process their jobs. This subset is usually defined either by the resource owners' policy (user is allowed to use such cluster), or by the user who requests some properties (library, software license, execution time limit, etc.) offered by some clusters or machines only. Also, the combination of both owners' and users' restrictions is possible. When one tries to create a new scheduling algorithm and compare it with current approaches such as EASY Backfilling [4], Conservative backfilling [5], or algorithms used in, e.g., PBSpro [6], LSF [7] or Moab [8], all such information and constraints are crucial, since they make the algorithm design much more complex. If omitted, resulting simulation may provide misleading or unrealistic results as we show in Section 6.

So far, we have been able to collect complex real-life data set from the Czech national Grid infrastructure MetaCentrum [9] that covers many previously mentioned issues such as machine parameters and supported properties, specific job requirements or machine failures. Using this complete data set [10] we were able to perform more realistic simulations. We have studied behavior of several objective functions that cover typical requirements such as the average job slowdown, the average response time, or the average wait time. We have compared schedule-based algorithms involving Local Search [1] which we have been developing for couple of years [11,12], as well as queue-based solutions such as FCFS or EASY and Conservative Backfilling. In our study, we have focused on two scenarios. The first (BASIC) scenario does not involve machine failures. Moreover, all jobs can be executed on any cluster (if enough CPUs are available), representing the typical amount of information available in the GWA or the PWA workloads. The second (EXTENDED) scenario uses additional information available in the MetaCentrum data set such as machine failures or additional cluster and job properties defining the job-to-cluster suitability (specific job requirements). As observed during the experiments (see Figure 2), the differences in the values of objective functions between these two scenarios are often large.

While the effects of machine failures on the cluster [13,14,15] or the Grid [13,16] performance are widely discussed, we are not aware of similar works that would also cover the effects of specific job requirements. Therefore, inspired by our own interesting results, we have decided to perform further analysis of existing workloads. When it was possible, we tried to recover additional information "hidden" in the available data covering both machine failure intervals and job requirements. When informations were insufficient we carefully generated synthetic machine failures using a statistical model. Once created, these "extended" workloads were compared through experiment with their original simpler versions. As expected, we have often discovered disproportion in the values of objective functions similar to the differences observed for the MetaCentrum data set. This supports our idea that scheduling algorithms should be evaluated using complete data sets.

The paper is organized as follows. First, we define the studied problem. Next, we discuss the PWA and GWA workloads, known failure traces and characteristics of considered workloads. The model used to create extended workloads is introduced and considered scheduling algorithms are described. We provide the detailed experimental evaluation with discussion of results and conclude our paper with a short summary.

2 Problem Description

In this section we describe the investigated job scheduling problems, starting with the simpler BASIC and followed by the EXTENDED problem. These problems are specified by characteristics of considered machines and jobs. We also define the optimization criteria considered for the evaluation of generated solutions.

2.1 BASIC Problem

The system is composed of one or more computer clusters and each cluster is composed of several machines. So far, we expect that all machines within one cluster have the same parameters. Those are the number of CPUs per machine and the CPU speed. All machines within a cluster use the Space Slicing processor allocation policy [17] which allows the parallel execution of several jobs at the cluster when the total amount of requested CPUs is less or equal to the number of CPUs of the cluster. Therefore, several machines within the same cluster can be co-allocated to process the given parallel job. On the other hand, machines belonging to different clusters can not be co-allocated to execute the same parallel job.

Job represents a user's application. Job may require one (sequential) or more CPUs (parallel). Also the arrival time and the job length are specified. There are no precedence constraints among jobs and we consider neither preemptions of the jobs nor migrations from one machine to another. When needed, the runtime estimates are precise (perfect) in this study.

2.2 EXTENDED Problem

This scenario extends the BASIC problem with some more features that are based on the characteristics of the MetaCentrum Grid environment. First of all, each cluster has additional parameters that closely specify its properties. These parameters typically describe the architecture of the underlying machines (Opteron, Xeon, ...), the available software licenses (Matlab, Gaussian, ...), the operating system (Debian, SUSE, ...), the list of queues allowed to use this cluster (each queue has a maximum time limit for the job execution, e.g., 2 hours, 24 hours, 1 month), the network interface parameters (10Gb/s, Infiniband, ...), the available file systems (nfs, afs, ...) or the cluster owner (Masaryk University, Charles University, ...). We expect that all the machines within one cluster have the same parameters.

Corresponding information is often used by the user to closely specify job's characteristics and requirements. Those are typically the time limit for the execution (given by the queue or user), the required machine architecture, the requested software licenses, the operating system, the network type or the file system. Users may also directly specify which cluster(s) is suitable for their jobs. In another words, by setting these requirements, user may prevent the job from running on some cluster(s). In real life, there are several reasons to do so. Some users strongly demand security and full control and they do not allow their jobs (and data) to use "suspicious" clusters which are not managed by their own organization. Others need special software such as Matlab or Gaussian which is not installed everywhere. Some clusters are dedicated for short jobs only (2 hours limit) and a user wanting more time is not allowed to use such cluster, and so on. All these requests are often combined together. In the EXTENDED problem all such requirements have to be included into the decision making process to satisfy all specific job's requirements. If no suitable machine is found, the job has to be cancelled. Clearly, the specific job requirements cannot be used when the corresponding cluster parameters are not known. Without them, consideration of "job-to-machine" suitability is irrelevant. Therefore, whenever the term *specific job requirements* is referenced in this paper, it means that both additional job and cluster parameters are applied, decreasing the number of suitable clusters for the job execution.

Finally, machine failures are considered in the EXTENDED scenario. It means that either one or more machines within a cluster are not available to execute jobs for some time period. Such failure may be caused by various reasons such as the power failure, the disk failure, the software upgrade, etc. However, we do not differentiate between them in this study. As a result of the failure, all jobs that have been — even partially — executed on such machine are immediately killed. Once the failure terminates, machine is restarted and becomes available for the job processing. Previously killed jobs are not resubmitted.

2.3 Evaluation Criteria

The quality of the generated solutions can be reflected by various types of optimization criteria. In both scenarios the following objective functions are considered: the avg. response time [17], the avg. slowdown [17] and the avg. wait time [18]. In addition, the total runtime of the scheduling algorithm is measured as a standard evaluation criteria. If machine failures are used we also count the total number of killed jobs. The avg. response time represents the average time a job spends in the system, i.e., the time from its submission to its termination. The avg. slowdown is the mean value of all jobs' slowdowns. Slowdown is the ratio of the actual response time of the job to the response time if executed without any waiting. Avg. wait time is the time that the job spends waiting before its execution starts. As pointed out by Feitelson et al. [17], the use of response time places more weight on long jobs and basically ignores if a short job waits few minutes, so it may not reflect users' notion of responsiveness. Slowdown reflects this situation, measuring the responsiveness of the system with respect

to the job length, i.e., jobs are completed within the time proportional to the job length. Wait time criterion supplies the slowdown and response time. Short wait times prevent the users from feeling that the scheduler "forgot about their jobs". All preceding objectives consider successfully executed jobs only, since killed jobs are not included. On the other hand, the total number of killed jobs is always monitored when machine failures are used. Finally, the total runtime of the scheduling algorithm measures the total CPU time needed by the scheduling algorithm to develop the solution for a given experiment.

3 Existing Sources of Workloads and Failure Traces

Two main publicly available sources of cluster and Grid workloads exist. Those are the Parallel Workloads Archive (PWA) [2] and the Grid Workloads Archive (GWA) [3]. There are two major differences between them. First of all, PWA maintains workloads coming from one site or cluster only (with few exceptions), while each workload in the GWA covers several sites. Second, the Grid Workloads Format (GWF) [19] is an extension to the Standard Workloads Format (SWF) [20] used in the PWA, reflecting some Grid specific job aspects. For example, each job in the GWF file has an identifier of the cluster where the job was executed. Moreover, the GWF format contains several fields to store specific job requirements. However, none of the six currently available traces uses them. These archives also often lack detailed and systematic description of the Grid or cluster resources, where the data were collected. Beside the real workloads, various models for generating synthetic workloads were proposed and implemented [21,22,23].

Traces of different kinds of failures related to the computer environment are collected in several archives such as the Repository of Availability Traces (RAT) [24] or the Computer Failure Data Repository (CFDR) [25]. For our purposes, the most suitable is the Failure Trace Archive (FTA) [13], that — among others — currently stores two Grid and cluster failure traces. Those are the Grid'5000 and the LANL traces. Remaining Grid or cluster related traces are either incomplete (PNNL) or were not yet converted from their original "raw" formats (EGEE, NERSC, HPC2, HPC4)[1]. FTA contains description of nodes but does not contain the information about jobs.

The complete MetaCentrum data set is publicly available at `http://www.fi.muni.cz/~xklusac/workload`. It contains trace of 103,620 jobs that includes specific job requirements as well as description of 14 clusters (806 CPUs) with the information about machine architecture, CPU speed, memory size and the supported properties. Also, the list of available queues including their priorities and associated time limits is provided. There is the trace of machine failures and the description of temporarily unavailable machines that were reserved or dedicated for special purposes. The average utilization of MetaCentrum varies per cluster with overall utilization being approximately 55%. In this work, we simulate neither reserved nor dedicated machines and we focus strictly on the

[1] In December 2009.

Table 1. Main characteristics of PWA, GWA, FTA and MetaCentrum archives

	PWA	**GWA**	**FTA**	**MetaCentrum**
job description	Yes	Yes	No	Yes
machine description	Partial	Partial	Yes	Yes
failures	No	No	Yes	Yes
specific job requirements	No	Partial	No	Yes

problem involving machine failures and specific job requirements. Therefore, the overall machine utilization has decreased to approximately 43% in our experiments[2].

Using these data sources we have selected three candidate workloads that have been used for the evaluation. Certainly the MetaCentrum workload was used as our base data set. Next, two more workloads were selected and carefully extended to obtain all information necessary for the EXTENDED problem. Methodologies used to generate such workloads are described in the next section. The SWF workload format does not contain information about job execution site, which is needed when generating extended workloads. Therefore, we were left with the GWA that contains six workloads now. However, three of them contain only sequential jobs, thus three candidates remained[3]: Grid'5000, DAS-2 and Sharcnet. Sadly, we had to eliminate Sharcnet since it does not provide enough information to generate the workload for the EXTENDED problem (see Section 4.2).

Grid'5000 is an experimental Grid platform consisting of 9 sites geographically distributed in France. Each site comprises one or several clusters, there are 15 clusters in total. The trace contains 1,020,195 jobs collected from May 2005 till November 2006. The total number of machines is not provided with the trace, therefore we had to reconstruct it from the job trace and information about machine failures available in the Grid'5000 failure trace. Then, we were able to determine the probable number of machines for each cluster. Totally, there has been approximately 1343 machines (2686 CPUs). Grid'5000 has a low average utilization being only 17%. On the other hand, there is a publicly available failure trace for Grid'5000 in the FTA, which is very convenient for our experiments. Sadly, all fields corresponding to specific job requirements are empty in the workload file.

DAS-2 (Distributed ASCI Supercomputer 2) workload trace comes from a wide-area Grid composed of 200 Dual Pentium-III nodes (400 CPUs). The Grid is built out of 5 clusters of workstations located at five Dutch Universities. Trace contains 1,124,772 jobs collected from February 2005 till December 2006. The workload has a very low utilization of approximately 10%. There is no failure trace available and the workload trace contains no specific job requirements.

[2] Machines dedicated or reserved for special purposes are considered as 100% utilized.
[3] If all jobs are sequential (non-parallel), then all scheduling algorithms considered in this paper follow more or less the FCFS approach.

Finally, Table 1 presents the main characteristics of PWA, GWA, FTA and MetaCentrum archives.

4 Extending the BASIC Problem

In this section we describe the main methods used to generate the synthetic machine failures and the specific job requirements. Using them, the Grid'5000 and the DAS-2 workloads were enriched towards the EXTENDED problem.

4.1 Machine Failures

Since both MetaCentrum and Grid'5000 data sets contain real machine failure traces, only the DAS-2 workload has been extended by the synthetic machine failures. First of all, the original DAS-2 workload was analyzed to determine the first time when each cluster was used by some job. As is shown in the Figure 1 (top left), only the DAS2/fs0 cluster was used from the beginning, four remaining clusters started to execute jobs approximately three months later. We have used this observation to generate four "initial cluster failures" that cover the time before the cluster was (probably) operational.

Next, the synthetic machine failures were generated. Five main parameters had to be determined. First of all, the total number of failures (F) has been established. Then, for each failure, four parameters have been chosen: failure duration, failure arrival time and the cluster and its machine that will exhibit this machine failure.

When solving these problems we were inspired by the model proposed by Zhang et al. in [14] and findings in [15,16,26]. We also used three available failure traces (MetaCentrum, Grid'5000, LANL) to get the necessary statistical data. As discussed in [15], failure rates are roughly proportional to the number of nodes (N) in the system. Therefore, the total number of failures (F) was computed as:

$$F = N \cdot D \cdot AFC \tag{1}$$

where D is the duration of DAS-2 workload in hours, and the AFC (Average Failure Count) is the average number of failures per machine per hour. While N and D were known, AFC had to be selected. Since we have no information concerning the real failure rates in DAS-2, we used known failure traces to compute the AFC values. Figure 1 (top middle) shows the AFC values for known failure traces. Grid'5000 shows suspiciously high AFC value, which is probably caused by the fact that some reported failures are rather "false alarms" than actual failures as discussed in [13]. In MetaCentrum, the AFC value is much more smaller while the low failure rates of LANL result in the lowest observed AFC. Since the large amount of failures in Grid'5000 is suspicious we have chosen LANL's and MetaCentrum's AFC values as two possible inputs into our failure generator. This resulted in two different failure traces for DAS-2. In the remaining text, DAS-2-L represents DAS-2 workload with failure trace generated using the LANL-based parameters while DAS-2-M represents solution based on the MetaCentrum parameters.

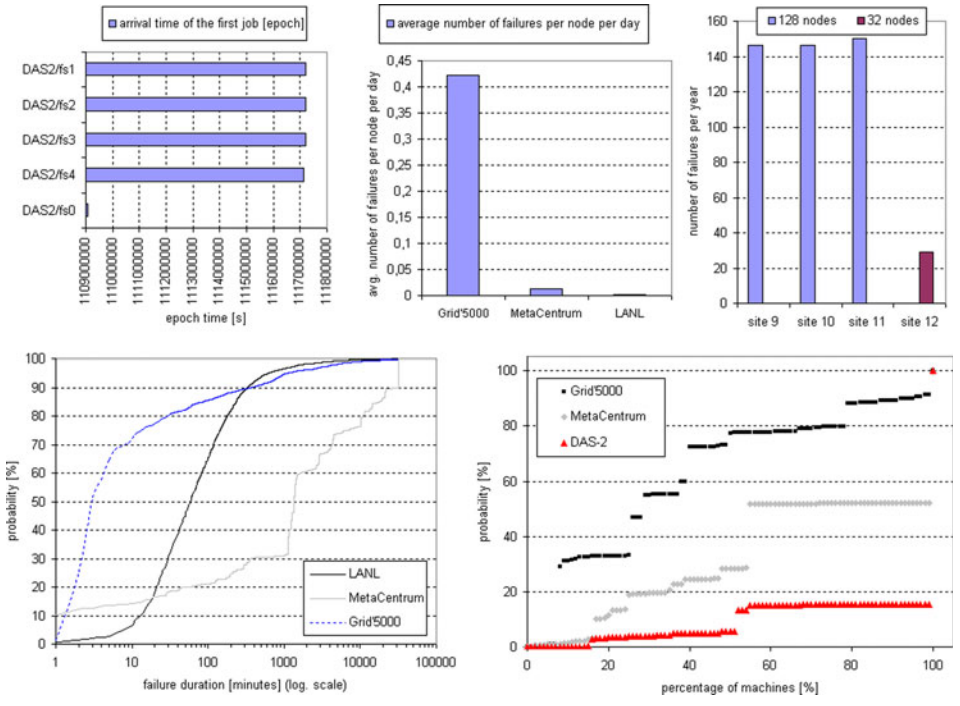


Fig. 1. The cluster start times in DAS-2 (top left), the average number of failures per node per day (*AFC*) (top middle), the number of failures per year in LANL (top right), the CDFs of failure durations for Grid'5000, LANL and MetaCentrum (bottom left) and the CDFs of "suitability distribution" of jobs onto clusters (bottom right)

Next, the remaining parameters were generated using the model [14] of Zhang et al. This involved the use of Weibull distribution [27] to generate the inter-arrival times between failures. Sahoo et al. [26] discussed that there are strong temporal correlations between failure arrivals. Using the model of Zhang et al., this behavior was simulated by including so called "failure bursts", i.e., multiple failures on one cluster appearing in (almost) the same time. Failure durations for DAS-2-L and DAS-2-M were generated using the Weibull distribution. Parameters of the distribution were selected by fitting the shape of Weibull Cumulative Distribution Function (CDF) [27] to the original CDFs of LANL and MetaCentrum failure durations that are shown in Figure 1 (bottom left)[4]. These CDFs represent the probability that the duration of machine failure will be less than or equal to x minutes.

The distribution of failures between clusters was done using the observations of LANL's failure distribution pattern shown in Figure 1 (top right) which has been closely discussed in [15]. Here, clusters with the same hardware and age

[4] The CDF of LANL is smoother since it was reconstructed from higher number of known failure durations. The x-axis is in log. scale.

have their failure rates roughly proportional to the number of machines within the cluster. This indicates that failure rates are not growing significantly faster than linearly with the cluster size [15]. Figure 1 (top right) shows this behavior for sites 9, 10, 11 and 12 in LANL. According to the available data, all DAS-2 clusters are based on the same hardware, therefore we have used the same linear distribution of failures in our failure generator.

Finally, the distribution of failures on the machines within one cluster was analyzed. Several authors show that such distribution is not uniform for a given cluster [26,28]. However, our own analysis of MetaCentrum failure trace showed that this is not always true. In MetaCentrum, some clusters had rather uniform failure distribution while for others it was highly unbalanced, showing significantly different shapes per each cluster. Since we have no reliable information about the type or shape of the desired distribution, we have decided to use simple uniform distribution in this case.

4.2 Specific Job Requirements

As far as we know there is no available model to simulate specific job requirements. Moreover, the only workload we are aware of that contains such information is the MetaCentrum workload. Our goal was to recreate such information for both DAS-2 and Grid'5000 workloads. Since it would be highly unreliable to simply transform known MetaCentrum pattern on different workloads, we have decided to use more realistic and conservative approach when establishing these requirements. Our approach is based on the analysis of the original DAS-2 and Grid'5000 workloads. In both of them each job contains identifier of the type (name) of the application that was used to execute the job as well as the identifier of the target cluster where it was finally executed [19]. Using this information, we could easily reveal the actual mapping of applications (jobs) on the clusters. To be more precise, we constructed a list of clusters where jobs having the same application identifier were executed. Next, during the simulation the application identifier is detected for each job and the corresponding clusters from the list are taken to be the only suitable execution sites for the job. Since we have no other information concerning job requirements, we used this mapping as the model of specific job requirements. Resulting CDFs based on such distributions are shown for the Grid'5000 and the DAS-2 workloads in Figure 1 (bottom right) together with the known distribution of MetaCentrum. Here, each CDF represents the probability that the job will be executable on at most $x\%$ of available machines. As it has been briefly mentioned in Section 3, this approach is not applicable for the Sharcnet workload, where the number of job identifiers is almost the same as the number of jobs in the workload. Thus, similar statistics did not make any sense and Sharcnet has not been used.

Table 2 summarizes the origins of all extensions of the original workloads. Presented generator of machine failures and specific job requirements can be downloaded at `http://www.fi.muni.cz/~xklusac/generator`

Table 2. Origin of machine failures and specific job requirements for the EXTENDED problem

	MetaCentrum	**Grid'5000**	**DAS-2**
machine failures	original data	original data	synthetic DAS-2-M
			synthetic DAS-2-L
specific job req.	original data	synthetic by workload analysis	synthetic by workload analysis

5 Scheduling Algorithms

Scheduling was performed by simulated centralized scheduler [1] that managed target clusters using different algorithms. We have used FCFS, EASY backfilling (EASY) [4] and Conservative backfilling (CONS) [5,29] optionally optimized with a Local Search (LS) algorithm [11,12]. EASY backfilling is an optimization of the FCFS algorithm, focusing on maximizing the system utilization. When the first (oldest) job in the queue cannot be scheduled because not enough processors are available, it calculates its earliest possible starting time using the runtime estimates of running jobs. Finally, it makes a reservation to run the job at this pre-computed time. Next, it scans the queue of waiting jobs and schedules immediately every job not interfering with the reservation of the first job. While EASY makes reservation for the first job only, Conservative backfilling makes the reservation for every queued job. These reservations represent an execution plan. We call this plan *the schedule* [30]. This schedule is updated whenever a new job arrives or some job completes its execution. Moreover, it allows us to apply advanced scheduling algorithms to optimize the schedule. This is the goal of the LS optimization procedure. LS maintains the schedule and optimizes its quality. New jobs are added to the schedule using CONS, i.e., they are placed to their earliest starting time. LS is run periodically and it consists of several iterations. In each iteration, random waiting job is selected and removed from the schedule and a new reservation is chosen randomly either on the same cluster or on a different one. Other reservations are updated with respect to this new assignment. Typically, when the original reservation is cancelled, later reservations can be shifted to the earlier start times. Analogically, new reservation can collide with existing reservations. If so, these are shifted to the later start times. Then, the *new schedule* is evaluated using the *weight* function W which is defined by Equation 2.

$$\begin{aligned} w_{sld} &= (sld_{previous} - sld_{new})/sld_{previous} \\ w_{wait} &= (wait_{previous} - wait_{new})/wait_{previous} \\ w_{resp} &= (resp_{previous} - resp_{new})/resp_{previous} \\ W &= w_{sld} + w_{wait} + w_{resp} \end{aligned} \tag{2}$$

W is a sum of three decision variables w_{sld}, w_{wait} and w_{resp} which are computed using the avg. job slowdown, avg. job wait time and avg. job response time of the previous and new schedule. They express the percentage increase or decrease in the quality of the new schedule with respect to the previous schedule. A positive value represents an improvement while a negative means that the new schedule represents a worse solution. Obviously, some correction is needed when the $wait_{previous}$ or $resp_{previous}$ is equal to zero but it is not presented to keep the code clear[5]. The final decision is based on the W value. If the W is greater than 0, then the new schedule is accepted, otherwise it is rejected and the schedule returns to the previous state. Iterations continue until the predefined number of iterations or the given time limit is reached. When applied, LS is executed every 5 minutes of simulation time. Here we were inspired by the actual setup of the PBSpro scheduler [6] used in the MetaCentrum which performs priority updates of jobs waiting in the queues with a similar periodicity. The maximum number of iterations is equal to the number of currently waiting jobs (schedule size) multiplied by 2. The maximum time limit was set to be 2 seconds which is usually enough to try all iterations. At the same time, it is still far less than the average job inter-arrival time of the densest DAS-2 trace (50 seconds). Since LS uses random function in each of its iteration, all experiments involving the LS algorithm have been repeated 10 times using different seeds, their results have been averaged and the standard deviation computed.

FCFS, EASY Backfilling and Conservative Backfilling are usually applied to schedule jobs on one cluster only. Since all our data sets involve several clusters, algorithms have been extended to allow scheduling over multiple clusters. This extension is very simple. FCFS, EASY and CONS simply check each cluster separately, finding the earliest possible reservation. If multiple choices to execute the job appear, the fastest available cluster is selected. If all available clusters have the same speed, the first choice is taken[6].

Next extension defines algorithms' reactions in case of a machine failure or restart. The simplest extension is made in FCFS. Here, machine failure or restart simply changes the set of CPUs to be used by the FCFS. Similar case applies for EASY, CONS and LS. However, machine failure may collide with existing reservations made by EASY, CONS or LS. If so, different actions are taken for EASY, CONS and LS. EASY checks whether the reservation of the first job is still valid. If not, it creates a new one. Since CONS and LS make reservation for every job it is more probable that collisions will appear. In our implementation, all jobs having reservations on the cluster where the machine failure occurred are re-backfilled using CONS. Other jobs' reservations are not changed since the total re-computation of all reservations for all clusters is very time consuming as we have have observed in our initial tests. If there are many machine failures, some highly parallel jobs may not be able to get a reservation, because there is not enough CPUs available in the system. If so, these jobs are canceled and removed from the queue since their huge wait times would distort the simulation

[5] By definition, slowdown is always greater or equal to 1.
[6] Clusters are ordered according to the total number of CPUs in descending order.

results. Jobs killed due to a machine failure are not resubmitted. Upon a machine restart, both FCFS and EASY try to use new CPUs immediately. CONS and LS behave somehow different since they have a reservation for every job at that moment. All reservations on the cluster where the machine restart appeared are recreated to utilize the restarted machine. Again, only the jobs having a reservation on such cluster are re-backfilled to minimize the algorithm's runtime. Reservations of jobs on remaining clusters are not changed. It may result in a temporally unbalanced distribution of jobs, since the re-backfilled jobs may not utilize all CPUs, especially if many machines restarted at the same moment. Such CPUs can be potentially suitable for jobs having reservations on different clusters. However, this imbalance is only temporal as new job arrivals or LS optimization will quickly saturate the available CPUs.

The inclusion of specific job requirements is very simple. All scheduling algorithms will allow job's execution on some cluster(s) if and only if the cluster(s) meets all specific job requirements.

6 Evaluation

All simulations were performed using the GridSim [31] based Alea simulator [32] on an Intel QuadCore 2.6 GHz machine with 2048MB of RAM. We have compared values of selected objective functions and the algorithms' runtime when the original (BASIC problem) and extended workloads (EXTENDED problem) have been used, respectively. As mentioned in Section 2, BASIC problem does not involve machine failures and specific job requirements while the EXTENDED does. In order to closely identify the effects of machine failures and specific job requirements on the values of objective functions, we have considered three different problems using the extended workloads. In EXT-FAIL only the machine failures are used and the specific job requirements are ignored. EXT-REQ represents the opposite problem, where the failures are ignored and only the specific job requirements are simulated. Finally, EXT-ALL uses both machine failures and specific job requirements. Using these setups, four different experiments were conducted for MetaCentrum and Grid'5000: BASIC, EXT-FAIL, EXT-REQ and EXT-ALL. Since DAS-2 has two variants of failure traces (DAS-2-L and DAS-2-M), there are six different experiments for the DAS-2 workload: BASIC, EXT-FAIL-L, EXT-FAIL-M, EXT-REQ, EXT-ALL-L and EXT-ALL-M, where "-L" or "-M" suffix specifies whether DAS-2-L or DAS-2-M failure trace has been used. Table 3 summarizes all data sets and problems we have considered in our experiments.

We start our discussion with the MetaCentrum workload where all information needed to simulate the EXTENDED problem were known, thus these results are the most reliable (see Figure 2). Next, we continue with the Grid'5000 (see Figure 3), where the EXTENDED problem was created using known data from the Failure Trace Archive (machine failures) and synthetically generated specific job requirements. Finally, Figure 5 presents results for the DAS-2 where additional data for the EXTENDED problem were generated synthetically. Resulting

Table 3. Overall summary of workloads, problems and performed experiments

	MetaCentrum	**Grid'5000**	**DAS-2**
BASIC	BASIC	BASIC	BASIC
EXTENDED	EXT-FAIL	EXT-FAIL	EXT-FAIL-L
			EXT-FAIL-M
	EXT-REQ	EXT-REQ	EXT-REQ
	EXT-ALL	EXT-ALL	EXT-ALL-L
			EXT-ALL-M

values of the avg. slowdown [17], the avg. response time [17] and the avg. wait time [18] are discussed for all experiments outlined in Table 3. For all data sets the total algorithm runtime follows the expectations — the more complex problem is considered and the more sophisticated algorithm is applied, the higher the algorithm runtime is. Machine failures often collide with existing reservations which have to be recreated. When also specific job requirements are used, checks to identify suitable clusters must be performed for each job, increasing the total runtime of all scheduling algorithms.

6.1 MetaCentrum Workload

Figure 2 shows the results for the MetaCentrum workload. As we can see the highest differences in the values of objective functions appear between BASIC and EXT-ALL experiments, which correspond with our expectations. In case of BASIC, the differences between all algorithms are not very large for all considered objectives. On the other hand, when the EXT-ALL problem is applied, large differences appear for all criteria. It is most significant in case of FCFS which generates the worst results among all applied algorithms. The values of FCFS are truncated for better visibility. EASY and CONS perform much better, while the best results are achieved by LS in most cases. Interesting results are related to the EXT-FAIL and EXT-REQ scenarios. As we can see, the inclusion of machine failures (EXT-FAIL) has usually a smaller effect than the inclusion of specific job requirements (EXT-REQ). Clearly, it is "easier" to deal with machine failures than with specific job requirements when the overall system utilization is not extreme. In case of a failure, the scheduler has usually other options where to execute the job. On the other hand, if the specific job requirements are taken into account other possibilities may not exist, and jobs with specific requests have to wait until the suitable machines become available.

The comparison of EASY and CONS is interesting as well. Many previous studies have tried to analyze their behavior [5,33,29,34]. A deep study of Feitelson [5] suggests that CONS is likely to produce better slowdown than EASY when precise runtime estimates are used and the system utilization is higher than approximately 50%. Similar behavior can be seen in the case of MetaCentrum, namely for EXT-REQ and EXT-ALL problems. Feitelson observed such

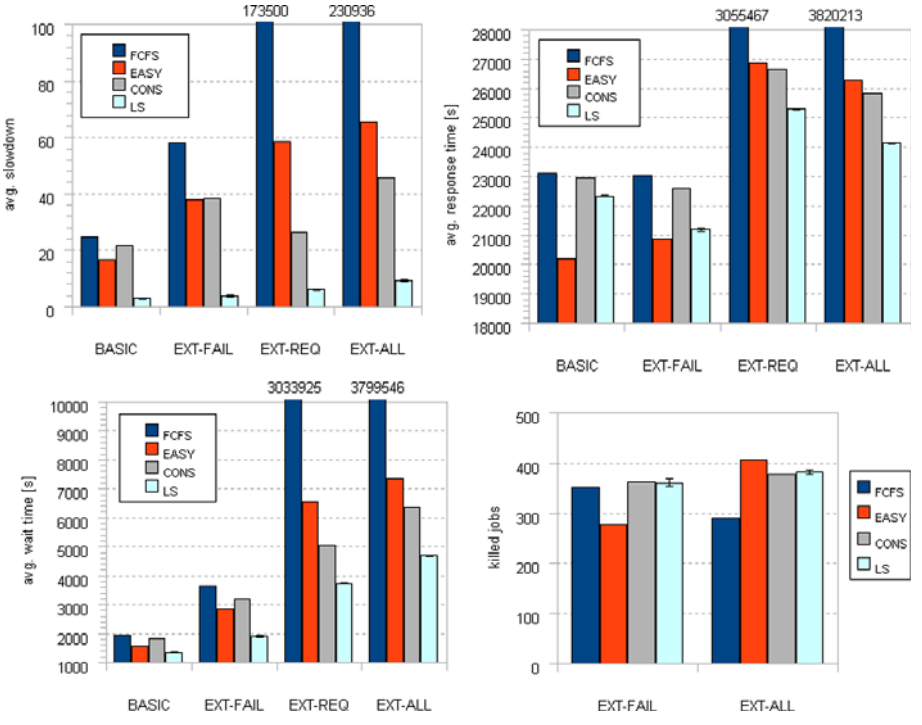


Fig. 2. Observed values of objective functions and the number of killed jobs for the MetaCentrum workload

large differences for workloads with at least 55-65% (Jann, Feitelson) or 85-95% (CTC) system utilization. For smaller system utilization, the performance of EASY and CONS was similar. However, the utilization of MetaCentrum is 43% on average. The reason for this behavior lies in the use of specific job requirements (EXT-REQ, EXT-ALL). As discussed, here jobs with specific requirements have to wait until the suitable machines become available. This in fact generates a higher system utilization on particular clusters, thus the benefits of CONS in this situation appear even for systems with a lower overall utilization. If no specific job requirements are used (BASIC, EXT-FAIL), CONS produces worse or equal results than EASY in all cases.

Feitelson [5] suggests that EASY should generate better response time than CONS. This is clearly recognizable for BASIC and EXT-FAIL problems, while CONS is slightly better for EXT-REQ and EXT-ALL, but the difference is very small. Periodical application of LS optimization routine always improve the solution with respect to the CONS and — with an exception of the avg. response time in BASIC and EXT-FAIL — LS generates the best results among all algorithms. The standard deviation of different LS executions is very small. Concerning the total number of killed jobs, there is no clear pattern indicating the best algorithm. To sum up, the use of specific job requirements and machine failures significantly influence the quality of generated solution. In case of MetaCentrum,

an experimental evaluation ignoring these features may be quite misleading. As was shown, the optimistic results for the BASIC problem are very far from those appearing when a more realistic EXT-ALL problem is considered.

6.2 Grid'5000 Workload

Figure 3 shows the results for the Grid'5000 workload. Similarly to the MetaCentrum workload, the highest differences appear between BASIC and EXT-REQ and EXT-ALL problems as can be seen in the case of the avg. slowdown and the avg. wait time. Again, due to the same reasons as before, the inclusion of specific job requirements (EXT-REQ) has a higher effect than the inclusion of machine failures (EXT-FAIL). The values of FCFS are often truncated for better visibility.

A closer attention is required when analyzing the average response time for EXT-FAIL and EXT-ALL problems in Grid'5000 shown in Figure 3 (top right). Initially, it is quite surprising that the average response time for EXT-FAIL and EXT-ALL is smaller than for the BASIC problem. The explanation is quite

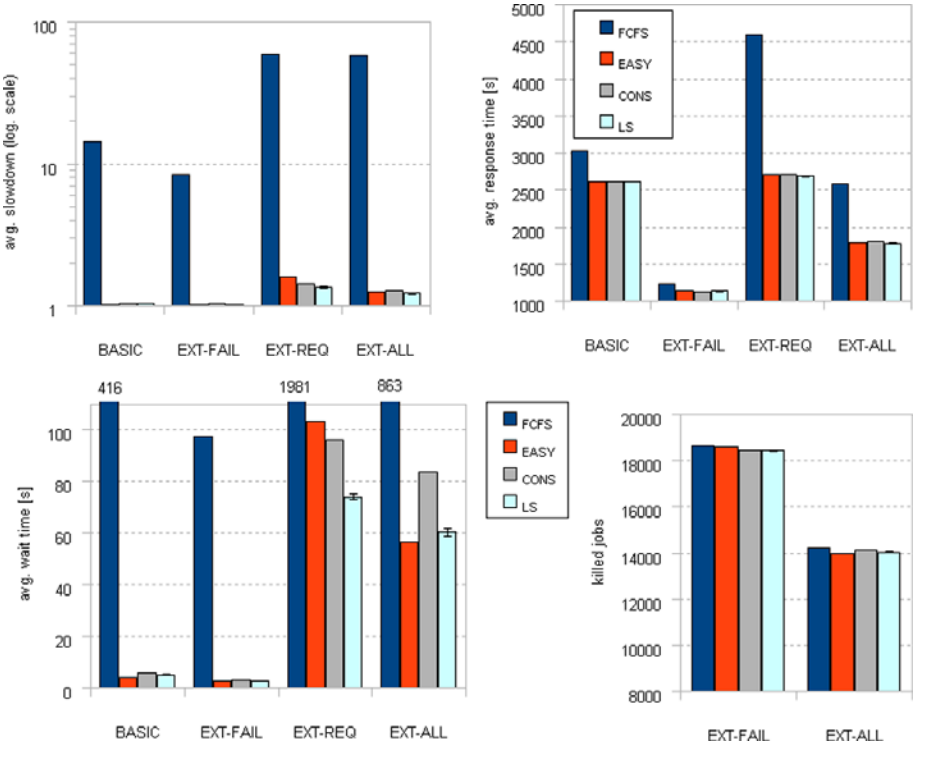


Fig. 3. Observed values of objective functions and the number of killed jobs for the Grid'5000 workload

straightforward. In our simulations, whenever some machine fails, all jobs being executed on this machine are killed immediately. Moreover, such jobs are not resubmitted. As mentioned in Section 4.1, the failure rate in Grid'5000 is very high causing many premature job terminations (see Figure 3, bottom right). Therefore, long jobs are more likely to be killed than the short ones. Our experiments confirmed this expectations. The average length of a killed job in EXT-FAIL (FCFS) has been 60,690 seconds. However, the average length of all jobs in Grid'5000 is just 2,608 seconds. It means that especially long jobs are being killed in this case. Therefore, whenever machine failures have been used (EXT-FAIL, EXT-ALL), the average response time has been smaller than for the BASIC and the EXT-REQ problems, since many long jobs have been killed and their long response times could not have been taken into account. The comparison of the avg. slowdown or the avg. wait time for BASIC and EXT-FAIL shows nearly no difference. Failures in Grid'5000 are usually very short as is shown in Figure 1 (bottom left). Therefore, they do not cause significant delays in job executions, although they appear very frequently. Moreover, the system utilization is very low (17%), so there is usually enough free CPUs to immediately start the execution of a newly incoming job.

As expected, FCFS did not perform well. In the Grid'5000 job trace, there are several jobs that request a large number of CPUs and only the largest clusters can be used. Since FCFS does not allow backfilling, smaller jobs in the queue have to wait until such large job has enough free CPUs to start its execution. It produces huge slowdowns for short jobs, although the overall utilization is only 17%. All remaining algorithms have been able to deal with such situations more efficiently, producing more or less similar solutions.

An interesting result is related to the Figure 3 (bottom right) showing the number of killed jobs. Here the total number of failed jobs for EXT-ALL is significantly lower than in the case of EXT-FAIL. This behavior is a combination of three factors and needs a closer explanation. First factor is related to the cluster selection process. If there are more suitable clusters to execute a given job then all scheduling algorithms applied in this paper select the fastest one. However, there are no information about clusters' speed in Grid'5000 and DAS-2, thus all clusters are considered to be equally fast. In such situation, all algorithms will choose the first suitable cluster (see Section 5), i.e., "the first fit" approach is applied. Since the Grid'5000 has a very small utilization (second factor) and clusters are always checked in a given order, most jobs are actually executed on the largest cluster. The third factor is the high failure rate in Grid'5000 (see Figure 1 top middle). The largest cluster exhibits 42% of all failures. When many jobs are executed on this single cluster, then the probability that machine failure will kill some job is rather high. For FCFS, there were 18668 killed jobs in the EXT-FAIL problem. 95.3% of them were killed at the largest cluster. Once the EXT-ALL problem is being solved, specific job requirements cause that some jobs have to be executed on different clusters. As a side effect, the number of failed jobs decreases, as observed in Figure 3 (bottom right). To conclude, the combination of the "first fit" cluster selection policy together with

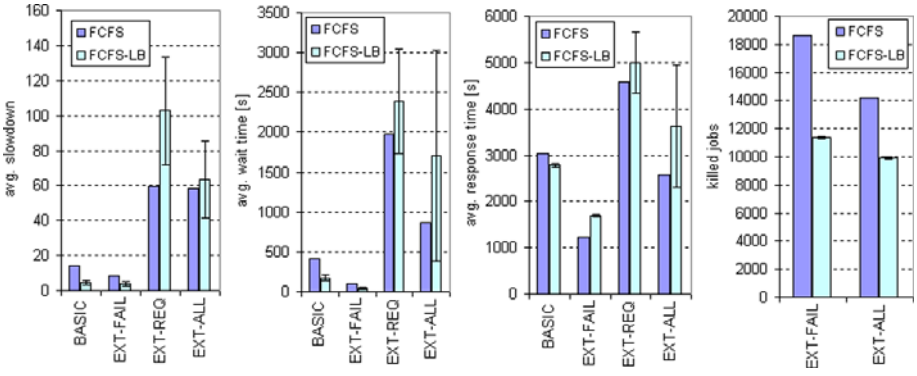


Fig. 4. Observed values of objective functions and the number of killed jobs for the FCFS and FCFS-LB in Grid'5000

high failure rate and low utilization may be very dangerous with respect to the number of killed jobs according to our observation. To prove this hypothesis we have developed a new version of the cluster selection policy, where — if multiple choices are available — a random cluster is selected using uniform distribution rather than the first one. We have used this policy in FCFS-LB (FCFS with Load Balancing) scheduling algorithm and compared this solution with the original "first fit" FCFS. The results are shown in Figure 4. All experiments involving the FCFS-LB algorithm have been repeated 10 times using different seeds and their results have been averaged and the standard deviation computed. Concerning the number of killed jobs (rightmost graph) we can see that FCFS-LB works much better than FCFS since jobs are uniformly spread over available clusters. Also the avg. slowdown and avg. wait time is slightly better for BASIC and EXT-FAIL. On the other hand, as soon as specific job requirements are considered (EXT-REQ, EXT-ALL) FCFS-LB produces worse results on average. Closer inspection shows, that the actual performance depends on the seed used in the random number generator, as can be seen from the large values of standard deviations. Clearly, simple solution such as FCFS-LB is not sufficient for more complex problems. We will try to fully understand this phenomena in the future since it is beyond the focus of this paper.

6.3 DAS-2 Workload

Figure 5 shows the results for the DAS-2 workload. DAS-2 uses artificially generated failure traces. To obtain more reliable results, all experiments involving machine failures have been repeated 10 times using different instances of failure traces. Each failure trace instance have been generated using a different seed. Finally, all results have been averaged and the standard deviation computed. As before, the highest differences in the values of objective functions appear between BASIC and EXT-ALL-L/EXT-ALL-M problems. For BASIC, no matter what scheduling algorithms is applied, the avg. slowdown, the avg. response time

and the avg. wait time are always nearly the same. Again, the poorest results are related to the application of FCFS. The disproportion between EXT-FAIL-L and EXT-FAIL-M observed for FCFS can be still reduced by any other more complex algorithm. Similarly to the MetaCentrum workload, LS generates the best results in all cases. As soon as EXT-ALL-L or EXT-ALL-M is applied, the differences between scheduling algorithms become more visible. Especially for the EXT-ALL-M, the use of more complex scheduling algorithms start to make sense. Although the absolute differences of selected objective functions between BASIC and EXT-ALL-M or EXT-ALL-L are not very large, still the application of machine failures and especially specific job requirements results in different behavior even for so lowly utilized system (10%) as the DAS-2 is. When the higher failure rate of the MetaCentrum workload is used to generate the failures (EXT-ALL-M) the resulting values of the avg. wait time and the avg. response time are worse than the corresponding values for the workload with LANL-based failures (EXT-ALL-L). In this case (EXT-FAIL-M, EXT-ALL-M), the performance of FCFS often highly oscillates as demonstrate the large values

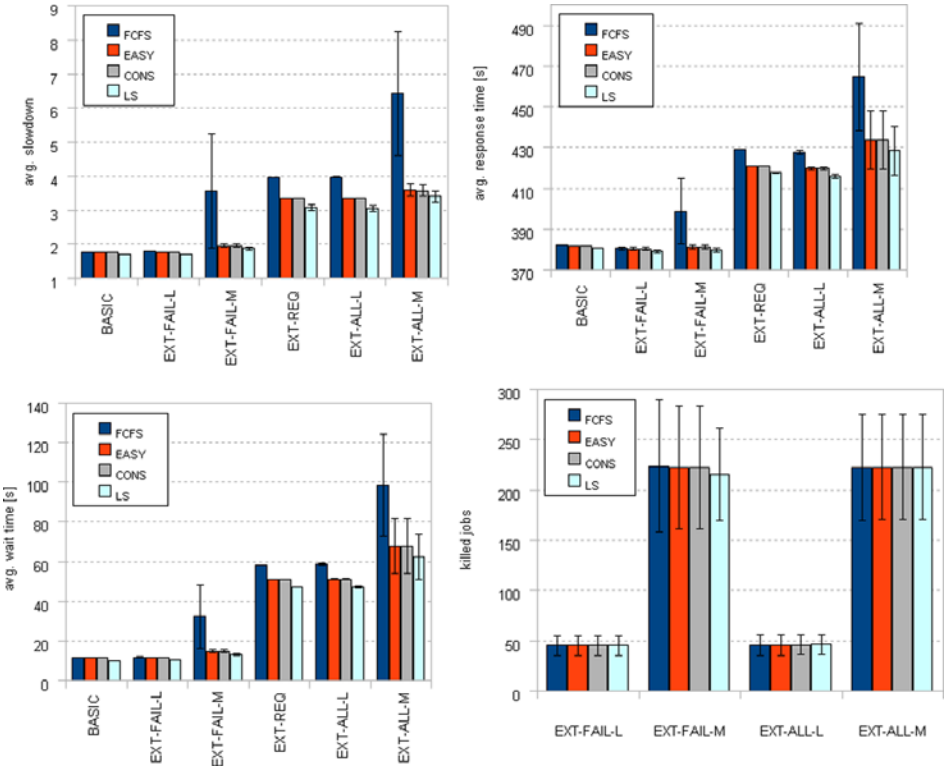


Fig. 5. Observed values of objective functions and the number of killed jobs for the DAS-2 workload

of standard deviation. FCFS is very sensitive to the used failure trace instance, while all remaining algorithms show quite stable performance. Also, the number of killed jobs is higher when the MetaCentrum-based failures are used which is expectable. Again, the use of specific job requirements (EXT-REQ) has usually higher effect than the use of machine failures only (EXT-FAIL-L, EXT-FAIL-M).

6.4 Summary

In this section we have shown that the use of complete data sets has significant influence on the quality of generated solutions. Similar patterns in the behavior of scheduling algorithms have been observed using three different data sets. First of all, the differences between studied algorithms are usually small for the BASIC problems, but the situation changes for the EXTENDED problems. Here, an application of more intelligent scheduling techniques may lead to significant improvements in the quality of generated solutions, especially when the system utilization is not very low. In such case, the optimization provided by Local Search (LS) algorithm may outperform all remaining algorithms as observed for the MetaCentrum data set. LS operates over the schedule of job reservations that is generated by CONS. When the system is lowly utilized, such schedule is often empty since jobs are executed immediately after their arrival. Then, LS has a little chance for optimization and its performance is very close to the original CONS algorithm as observed in Grid'5000 and DAS-2 cases. In addition, specific job requirements may have higher impact than machine failures. So far, the differences between the EASY and CONS solutions were likely to appear in systems with high utilization [5]. As observed in the MetaCentrum experiment, the application of specific job requirements can significantly decrease the threshold of the system utilization when the differences between algorithms are likely to appear. The inclusion of machine failures may have severe effect on the values of objective functions as observed mainly in Grid'5000 having a very high failure rate. In this case, another objectives such as the total number of killed jobs must be taken into account to explain otherwise "confusing" results. Moreover, low utilization, high failure rates combined with scheduler's selection policies could bring unexpected problems such as high numbers of killed jobs ("first-fit" job allocation). Trying to solve this problem using some form of load balancing may help, but other objectives can be easily degraded due to highly unstable performance as observed for FCFS-LB. Here, immediate solutions such as the simple load balancing do not work very well, since many other factors interact together. These observations support our idea that complete workload traces should be collected, published and used by the scientific community in the future. They will allow to test more realistic scenarios and they will help to understand the complicated behavior of real, complex systems.

7 Conclusion

Based on the real-life data from the Czech Grid MetaCentrum, we have demonstrated that machine failures and specific job requirements significantly affect the performance of various scheduling algorithms. Since the workloads in current archives miss to capture these features we have carefully extended selected existing workloads to show that they may exhibit similar behavior. Clearly, complete and "rich" data sets influence the algorithms' behavior and causes significant differences in the values of objective functions with respect to the simple versions of the problems. As far as we know, specific job requirements have not been used in the context of Grid and cluster scheduling so far. We have shown that they should not be underestimated and their effects should be closely studied in the future. We suggest, that beside the common workloads from the GWA and the PWA, also the complete ones should be collected, published and applied to evaluate existing and newly proposed algorithms under harder conditions. As it was presented, existing base workloads may not clearly demonstrate the differences between trivial and advanced scheduling algorithms. When possible, detailed and standardized description of the original cluster and Grid environment should be provided as well, to assure that simulations will use correct setups. As a first step we provide the complex MetaCentrum data set for further open research.

Acknowledgments. We appreciate the gracious support of the Ministry of Education, Youth and Sports of the Czech Republic under the research intent No. 0021622419. We also appreciate the help of MetaCentrum, graciously supported by the research intent MSM6383917201. MetaCentrum team provided us the workload data and help us to interpret them correctly. The DAS-2 workload was kindly provided by the Advanced School for Computing and Imaging, the owner of the DAS-2 system, while the Grid'5000 workload and failure traces were kindly provided by the Grid'5000 team. We also appreciate the LANL failure traces graciously provided by the LANL team.

References

1. Xhafa, F., Abraham, A.: Computational models and heuristic methods for grid scheduling problems. Future Generation Computer Systems 26(4), 608–621 (2010)
2. Feitelson, D.G.: Parallel workloads archive (PWA), `http://www.cs.huji.ac.il/labs/parallel/workload/`
3. Epema, D., Anoep, S., Dumitrescu, C., Iosup, A., Jan, M., Li, H., Wolters, L.: Grid workloads archive (GWA), `http://gwa.ewi.tudelft.nl/pmwiki/`
4. Skovira, J., Chan, W., Zhou, H., Lifka, D.: The EASY - LoadLeveler API project. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1996 and JSSPP 1996. LNCS, vol. 1162, pp. 41–47. Springer, Heidelberg (1996)
5. Feitelson, D.G.: Experimental analysis of the root causes of performance evaluation results: A backfilling case study. IEEE Transactions on Parallel and Distributed Systems 16(2), 175–182 (2005)
6. Jones, J.P.: PBS Professional 7, administrator guide. Altair (2005)

7. Xu, M.Q.: Effective metacomputing using LSF multicluster. In: CCGRID 2001: Proceedings of the 1st International Symposium on Cluster Computing and the Grid, pp. 100–105. IEEE, Los Alamitos (2001)
8. Cluster Resources: Moab workload manager administrator's guide, version 5.3 (2010), http://www.clusterresources.com/products/mwm/docs/
9. MetaCentrum, http://meta.cesnet.cz/
10. Klusáček, D., Rudová, H.: Complex real-life data sets in Grid simulations (abstract). In: Cracow Grid Workshop 2009 Abstracts (CGW 2009), Cracow, Poland (2009)
11. Klusáček, D., Rudová, H.: Efficient grid scheduling through the incremental schedule-based approach. Computational Intelligence: An International Journal (to appear 2010)
12. Klusáček, D., Rudová, H., Baraglia, R., Pasquali, M., Capannini, G.: Comparison of multi-criteria scheduling techniques. In: Grid Computing Achievements and Prospects, pp. 173–184. Springer, Heidelberg (2008)
13. Kondo, D., Javadi, B., Iosup, A., Epema, D.: The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems. Technical Report 00433523, INRIA (2009)
14. Zhang, Y., Squillante, M.S., Sivasubramaniam, A., Sahoo, R.K.: Performance implications of failures in large-scale cluster scheduling. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2004. LNCS, vol. 3277, pp. 233–252. Springer, Heidelberg (2005)
15. Schroeder, B., Gibson, G.A.: A large-scale study of failures in high-performance computing systems. In: DSN 2006: Proceedings of the International Conference on Dependable Systems and Networks, pp. 249–258. IEEE Computer Society, Los Alamitos (2006)
16. Iosup, A., Jan, M., Sonmez, O., Epema, D.H.J.: On the dynamic resource availability in grids. In: GRID 2007: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing, pp. 26–33. IEEE Computer Society, Los Alamitos (2007)
17. Feitelson, D.G., Rudolph, L., Schwiegelshohn, U., Sevcik, K.C., Wong, P.: Theory and practice in parallel job scheduling. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1997 and JSSPP 1997. LNCS, vol. 1291, pp. 1–34. Springer, Heidelberg (1997)
18. Ernemann, C., Hamscher, V., Yahyapour, R.: Benefits of global Grid computing for job scheduling. In: GRID 2004: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, pp. 374–379. IEEE, Los Alamitos (2004)
19. Iosup, A., Li, H., Jan, M., Anoep, S., Dumitrescu, C., Wolters, L., Epema, D.H.J.: The grid workloads archive. Future Generation Computer Systems 24(7), 672–686 (2008)
20. Chapin, S.J., Cirne, W., Feitelson, D.G., Jones, J.P., Leutenegger, S.T., Schwiegelshohn, U., Smith, W., Talby, D.: Benchmarks and standards for the evaluation of parallel job schedulers. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 1999, IPPS-WS 1999 and SPDP-WS 1999. LNCS, vol. 1659, pp. 67–90. Springer, Heidelberg (1999)
21. Tsafrir, D., Etsion, Y., Feitelson, D.G.: Modeling user runtime estimates. In: Feitelson, D.G., Frachtenberg, E., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2005. LNCS, vol. 3834, pp. 1–35. Springer, Heidelberg (2005)
22. Lublin, U., Feitelson, D.G.: The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. Journal of Parallel and Distributed Computing 63(11), 1105–1122 (2003)

23. Feitelson, D.G., Rudolph, L.: Metrics and benchmarking for parallel job scheduling. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1998, SPDP-WS 1998, and JSSPP 1998. LNCS, vol. 1459, pp. 1–24. Springer, Heidelberg (1998)
24. Repository of availability traces (RAT), `http://www.cs.illinois.edu/~pbg/availability/`
25. The computer failure data repository (CFDR), `http://cfdr.usenix.org/`
26. Sahoo, R.K., Sivasubramaniam, A., Squillante, M.S., Zhang, Y.: Failure data analysis of a large-scale heterogeneous server environment. In: DSN 2004: Proceedings of the 2004 International Conference on Dependable Systems and Networks, pp. 772–784. IEEE Computer Society, Los Alamitos (2004)
27. Johnson, N.L., Kotz, S., Balakrishnan, N.: Continuous Univariate Distributions, 2nd edn., vol. 1. Wiley-Interscience, Hoboken (1994)
28. Heath, T., Martin, R.P., Nguyen, T.D.: Improving cluster availability using workstation validation. ACM SIGMETRICS Performance Evaluation Review 30(1), 217–227 (2002)
29. Srinivasan, S., Kettimuthu, R., Subramani, V., Sadayappan, P.: Selective reservation strategies for backfill job scheduling. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2002. LNCS, vol. 2537, pp. 55–71. Springer, Heidelberg (2002)
30. Hovestadt, M., Kao, O., Keller, A., Streit, A.: Scheduling in HPC resource management systems: Queueing vs. planning. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2003. LNCS, vol. 2862, pp. 1–20. Springer, Heidelberg (2003)
31. Sulistio, A., Cibej, U., Venugopal, S., Robic, B., Buyya, R.: A toolkit for modelling and simulating data Grids: an extension to GridSim. Concurrency and Computation: Practice & Experience 20(13), 1591–1609 (2008)
32. Klusáček, D., Rudová, H.: Alea 2 – job scheduling simulator. In: Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools 2010), ICST (2010)
33. Mu'alem, A.W., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. IEEE Transactions on Parallel and Distributed Systems 12(6), 529–543 (2001)
34. Krallmann, J., Schwiegelshohn, U., Yahyapour, R.: On the design and evaluation of job scheduling algorithms. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 1999, IPPS-WS 1999, and SPDP-WS 1999. LNCS, vol. 1659, pp. 17–42. Springer, Heidelberg (1999)

Hierarchical Scheduling of DAG Structured Computations on Manycore Processors with Dynamic Thread Grouping*

Yinglong Xia[1], Viktor K. Prasanna[1,2] and James Li[2]

[1] Department of Computer Science, University of Southern California, Los Angeles, CA 90089, U.S.A.
[2] Ming Hsieh Department of Electrical Engineering, University of Southern California, Los Angeles, CA 90089, U.S.A.
{yinglonx,prasanna,jamesyli}@usc.edu

Abstract. Many computational solutions can be expressed as directed acyclic graphs (DAGs) with weighted nodes. In parallel computing, scheduling such DAGs onto manycore processors remains a fundamental challenge, since synchronization across dozens of threads and preserving precedence constraints can dramatically degrade the performance. In order to improve scheduling performance on manycore processors, we propose a hierarchical scheduling method with dynamic thread grouping, which schedules DAG structured computations at three different levels. At the top level, a supermanager separates threads into groups, each consisting of a manager thread and several worker threads. The supermanager dynamically merges and partitions the groups to adapt the scheduler to the input task dependency graphs. Through group merging and partitioning, the proposed scheduler can dynamically adjust to become a centralized scheduler, a distributed scheduler or somewhere in between, depending on the input graph. At the group level, managers collaboratively schedule tasks for their workers. At the within-group level, workers perform self-scheduling within their respective groups and execute tasks. We evaluate the proposed scheduler on the Sun UltraSPARC T2 (Niagara 2) platform that supports up to 64 hardware threads. With respect to various input task dependency graphs, the proposed scheduler exhibits superior performance when compared with other various baseline methods, including typical centralized and distributed schedulers.

Keywords: Manycore processor, hierarchical scheduling, thread grouping.

1 Introduction

Given a program, we can represent the program as a *directed acyclic graph* (DAG) with weighted nodes, in which the nodes represent code segments, and

* This research was partially supported by the National Science Foundation under grant number CNS-0613376. NSF equipment grant CNS-0454407 is gratefully acknowledged.

E. Frachtenberg and U. Schwiegelshohn (Eds.): JSSPP 2010, LNCS 6253, pp. 154–174, 2010.

edges represent dependencies among the segments. An edge exists from node v to node $\tilde{v}$ if the output from the code segment performed at v is an input to the code segment at $\tilde{v}$. The weight of a node represents the (estimated) execution time of the corresponding code segment. Such a DAG is called a *task dependency graph*, and the computations that can be represented as task dependency graphs are called *DAG structured computations* [1,2]. The *objective* of task scheduling for DAG structured computations on manycore processors is to minimize the overall execution time by proper allocation of the tasks to concurrent threads, while preserving the precedence constraints among the tasks [2,3].

Scheduling DAG structured computations on manycore processors is a fundamental challenge in parallel computing nowadays. The trend in architecture design is to integrate more and more cores onto a single chip to achieve higher performance. Such architectures are known as manycore processors. Examples of existing manycore processors include the Sun UltraSPARC T1 (Niagara) and T2 (Niagara 2), which support up to 32 and 64 concurrent threads, respectively [4]. The Nvidia Tesla and Tilera TILE64 are also available. More manycore processors are emerging soon, such as the Sun Rainbow Falls, IBM Cyclops64 and Intel Larrabee [5]. Such processors are more interested in how many tasks from a DAG can be completed efficiently over a period of time rather than how quickly an individual task can be completed.

Our contributions in this paper include: (a) We propose a hierarchical scheduling method which schedules DAG structured computations at three different levels on manycore systems. (b) We propose a dynamic thread grouping technique to merge or partition the thread groups at run time, so that the proposed scheduler can dynamically adjust to become a centralized scheduler, a distributed scheduler or somewhere in between, depending on the input graph. (c) We implement the hierarchical scheduling method on the Sun UltraSPARC T2 (Niagara 2) platform. (d) We conduct extensive experiments to validate the proposed method.

The rest of the paper is organized as follows: In Section 2, we review the background and related work. Section 3 presents the hierarchical scheduling scheme. We illustrate experimental results in Section 4 and address the future research in Section 5.

2 Background and Related Work

In this paper, the input to task scheduling is a directed acyclic graph (DAG), where each node represents a task and each edge corresponds to precedence constraints among the tasks. Each task in the DAG is associated with a *weight*, which is the estimated execution time of the task. A task can begin execution only if all of its predecessors have been completed [6]. The task scheduling problem is to map the tasks to the threads in order to minimize the overall execution time on parallel computing systems. Task scheduling is in general an *NP-complete* problem [7,8]. We consider scheduling an arbitrary DAG with given task weights and decide the mapping and scheduling of tasks on-the-fly. The goal of such

dynamic scheduling includes not only the minimization of the overall execution time, but also the minimization of the scheduling overhead [2].

The scheduling problem has been extensively studied for several decades [1,9,2,10]. Early algorithms optimized scheduling with respect to the specific structure of task dependency graphs [11], such as a tree or a fork-join graph. In general, however, programs come in a variety of structures [2]. Karamcheti and Chien studied hierarchical load balancing framework for multithreaded computations for employing various scheduling policies for a system [12]. Recent research on scheduling DAGs includes [13] where the authors studied the problem of scheduling more than one DAG simultaneously onto a set of heterogeneous resources, and [1] where Ahmad proposed a game theory based scheduler on multicore processors for minimizing energy consumption. Dongarra *et al.* proposed dynamic schedulers optimized for some linear algebra problems on general-purpose multicore processors [10]. Scheduling techniques have been proposed by several emerging programming systems such as Cilk [14], Intel Threading Building Blocks (TBB) [15], OpenMP [16], Charm++ [17] and MPI micro-tasking [18], etc. All these systems rely on a set of extensions to common imperative programming languages, and involve a compilation stage and runtime libraries. These systems are not optimized specifically for scheduling DAGs on manycore processors. For example, Dongarra *et al.* showed that Cilk is not efficient for scheduling workloads in dense linear algebra problems on multicore platforms [19]. In contrast with these systems, we focus on scheduling for DAGs on manycore processors.

To design an efficient scheduler we must take into account the architectural characteristics of processors. Almost all the existing manycore processors have relatively simple cores, compared with general-purpose multicore processors, e.g., AMD Opteron and Intel Xeon. For example, the pipeline of the UltraSPARC T2 does not support out of order (OoO) execution and therefore results in a longer delay. However, the fast context switch of such processors overlaps such delays with the execution of another thread. For this reason, the UltraSPARC generally shows higher throughput when enough parallel tasks are available [4].

Directly utilizing traditional scheduling methods such as centralized or distributed scheduling can degrade the performance of DAG structured computations on manycore processors. Centralized scheduling has a single thread to allocate tasks, which may not be able to serve the rest of the threads in time. This leads to starvation of some threads, especially when the tasks can be completed quickly. On the other hand, distributed scheduling requires many threads to schedule tasks. This limits the resources for task execution. In addition, many schedulers accessing shared variables can result in costly synchronization overhead. Therefore, an efficient scheduling method on manycore processors must be able to adapt itself to input task dependency graphs. To the best of our knowledge, no scheduling algorithm for DAG structured computations has been proposed specifically on manycore processors such as the UltraSPARC T2.

3 Hierarchical Scheduling

3.1 Organization

The input graph is represented by a list called the *global task list* (GL). Figure 1(a) shows a portion of the task dependency graph. Figure 1(b) shows the corresponding part of the GL. As shown in Figure 1(c), each element in the GL consists of task ID, dependency degree, task weight, successors and the task meta data (e.g. application specific parameters). The *task ID* is the unique identity of a task. The *dependency degree* of a task is initially set as the number of incoming edges of the task. During the scheduling process, we decrease the dependency degree of a task once a predecessor of the task is processed. The *task weight* is the estimated execution time of the task. We keep the task IDs of the *successors* along with each task to preserve the precedence constraints of the task dependency graph. When we process a task T_i, we can locate its successors directly using the successor IDs, instead of traversing the entire list. In each element, we have *task meta data*, such as the task type and pointers to the data buffer of the task, etc. The GL is shared by all the threads.

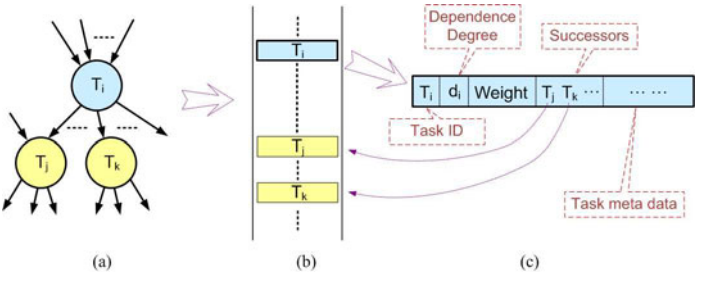


Fig. 1. (a) A portion of a task dependency graph. (b) The corresponding representation of the global task list (GL). (c) The data of element T_i in the GL.

We illustrate the components of the hierarchical scheduler in Figure 2. The boxes with rounded corners represent thread groups. Each group consists of a *manager* thread and several *worker* threads. The manager in Group_0 is also the *supermanager*. The components inside of a box are private to the group; while the components out of the boxes are shared by all groups.

The *global ready list* (GRL) in Figure 2 stores the IDs of tasks with dependency degree equal to 0. These tasks are ready to be executed. During the scheduling process, a task is put into this list by a manager thread once the dependency degree of the task becomes to 0.

The *local ready list* (LRL) in each group stores the IDs of tasks allocated to the group by the manager of the group. The workers in the group fetch tasks from LRL for execution. Each LRL is associated with a *workload indicator* (WI) to record the overall workload of the tasks currently in the LRL. Once a task is inserted into (or fetched from) the LRL, the indicator is updated.

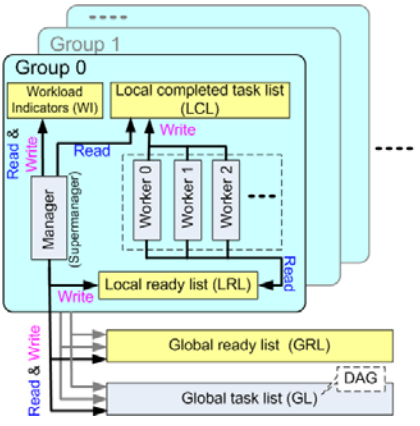


Fig. 2. Components of the hierarchical scheduler

The *local completed task list* (LCL) in each group stores the IDs of tasks completed by a worker thread in the group. The list is read by the manager thread in the group for decreasing the dependency degree of the successors of the tasks in the list.

The arrows in Figure 2 illustrate how each thread accesses a component (read or write). As we can see, GL and GRL are shared by all the managers for both read and write. For each group, the LRL is write-only for the manager and read-only for the workers; while LCL is write-only for the workers and read-only for the manager. WI is local to the manager in the respective group only.

3.2 Dynamic Thread Grouping

The scheduler organization shown in Figure 2 supports dynamic thread grouping, which means that the number of threads in a group can be adjusted at runtime. We adjust groups by either merging two groups or partitioning a group. The proposed organization ensures efficient group merging and partitioning.

Figure 3(a) illustrates the merging of Group_i and Group_j, $i < j$. The two groups are merged by converting all threads of Group_j into the workers of Group_i and merging WIs, LCLs and LRLs accordingly. Converting threads of Group_j into the workers of Group_i is straightforward: $Manager_j$ stops allocating tasks to Group_j, but performs self-scheduling as a worker thread. Then, all the threads in Group_j access tasks from the merged LRL and LCL. To combine WI_i and WI_j, we add the value of WI_j to WI_i. Although WI_j is not used after merging, we still keep it updated for the sake of possible future group partitioning. Merging the lists i.e. LCLs and LRLs is efficient. Note that both LCL and LRL are circular lists, each having a head and a tail pointer to indicate the first and last tasks stored in the list, respectively. Figure 3(b) illustrates the approach to merge two circular lists. We need to update two links only, i.e. the bold arrows shown in Figure 3(b). None of the tasks stored in the lists are moved or duplicated. The

head and tail of the merged list are $Head_i$ and $Tail_j$, respectively. Note that two merged groups can be further merged into a larger group.

We summarize the procedure in Algorithm 1. Since the queues and weight indicators are shared by several threads, locks must be used to avoid concurrent write. For example, we lock LRL_i and LRL_j immediately before Line 1 and unlock them after Line 3. Algorithm 2 does not explicitly assign the threads in Group_i and Group_j to Group_k, since this algorithm is executed only by the supermanager. Each thread dynamically updates its group information and decides if it should be a manager or worker (see Algorithm 2).

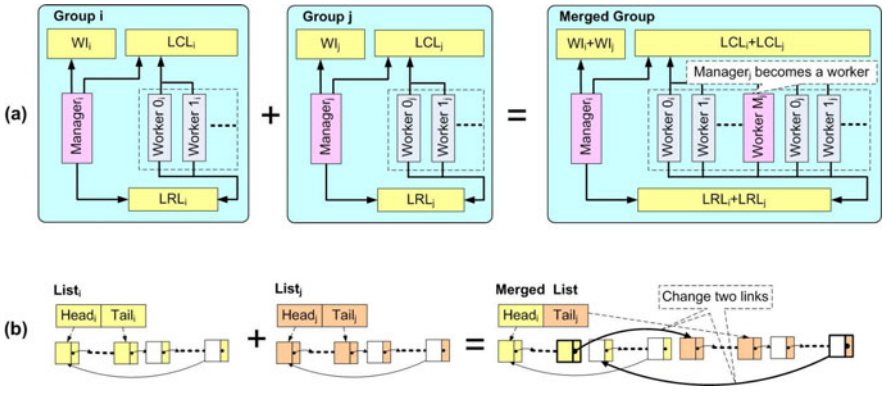


Fig. 3. (a) Merge Group_i and Group_j. (b) Merge circular lists $List_i$ and $List_j$. The head (tail) points to the first (last) tasks stored in the list. The blank elements have no task stored yet.

Algorithm 1. Group merge

Input: Group_i and Group_j.
Output: $\text{Group}_k = \text{Group}_i + \text{Group}_j$

{Merge LRL_i and LRL_j}
1: Let $LRL_j.Head.Predecessor$ points to $LRL_i.Tail.Successor$
2: Let $LRL_i.Tail.Successor$ points to $LRL_j.Head$
3: $LRL_k.Head = LRL_i.Head$, $LRL_k.Tail = LRL_j.Tail$

{Merge LCL_i and LCL_j}
4: Let $LCL_j.Head.Predecessor$ points to $LCL_i.Tail.Successor$
5: Let $LCL_i.Tail.Successor$ points to $LCL_j.Head$
6: $LCL_k.Head = LCL_i.Head$, $LCL_k.Tail = LCL_j.Tail$

{Merge WI_i and WI_j}
7: $WI_k = WI_i + WI_j$

Group_i and Group_j can be restored from the merged group by partitioning. As a reverse process of group merging, group partitioning is also straightforward and efficient. Due to space limitations, we do not elaborate it here. Group merging

and partitioning can be used for groups with an arbitrary number of threads. We assume the number of threads per group is a power of two hereinafter for the sake of simplicity.

3.3 Hierarchical Scheduling

Using the proposed data organization, we schedule a given DAG structured computation at three levels. The top level is called the *meta-level*, where we have a supermanager to control group merging/partitioning. At this level, we are not scheduling tasks directly, but reconfiguring the scheduler according to the characteristics of the input tasks. Such a process is called *meta-scheduling*. The supermanager is hosted along with the manager of Group_0 by Thread_0. Note that $Manager_0$ can never become a worker as discussed in Section 3.2.

The mediate level is called the *group level*, where the manager in each group collaborates with each other and allocates tasks for the workers in the group. The purpose of collaborating between managers is to improve the load balance across the groups. Specifically, the managers ensure that the workload in the local ready lists is roughly equal for all groups. A manager is hosted by the first thread in a group.

The bottom level is called the *within-group level*, where the workers in each group perform self-scheduling. That is, once a worker finishes a task execution and updates LCL in its group, it fetches a new task, if any, from LRL immediately. Self-scheduling keeps all workers busy, unless the LRL is empty. Each worker is hosted by a separate thread.

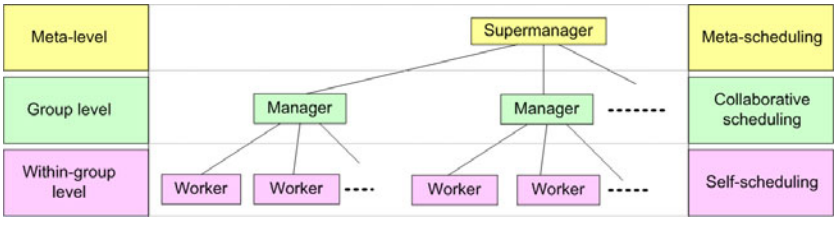


Fig. 4. The hierarchical relationship between the supermanager, managers and workers, and the corresponding scheduling schemes

The hierarchical scheduler behaves between centralized and distributed schedulers, so that it can adapt to the input task graph. Note that each group consists of a manager thread and several worker threads. When all the groups are merged into a single group, the proposed method becomes a centralized scheduler; when multiple groups exist, the proposed method behaves as a distributed scheduler.

3.4 Scheduling Algorithm and Analysis

We propose a sample implementation of the hierarchical scheduler presented in Section 3.3. Based on the organization shown in Section 3.1, we use the following

notations to describe the implementation: Assume there are P threads, each bound to a core. The threads are divided into groups consisting of a manager and several workers. GL and GRL denote the global task list and global ready list, respectively. LRL_i and LCL_i denote the local ready list and local completed task list of Group$_i$, $0 \leq i < P$. d_T and w_T represent the dependency degree and the weight of task T, respectively. WI_i is the workload indicator of Thread$_i$. Parameters δ_M, δ_+ and δ_- are given thresholds. The boxes show the statements that access variables shared by all groups.

Algorithm 2 illustrates the framework of the hierarchical scheduler. In Lines 1-3, thread groups are initialized, each with a manager and a worker, along with a set of ready-to-execute tasks stored in LRL_j, where the overall task weight is recorded in WI_j. A boolean flag f_{exit} in Line 3 notifies if the threads can exit the scheduling iteration (Lines 5-15). *rank* controls the size of groups: Increasing *rank* leads to merging of two adjacent groups; while decreasing *rank* leads to partitioning of current groups. $rank = 1$ corresponds to the minimum group size i.e. two threads per group. Thus, we have $1 \leq rank \leq \log P$. The group size Q is therefore given by:

$$Q = \frac{P}{2^{\log P - rank}} = 2^{rank} \tag{1}$$

Line 4 in Algorithm 2 starts all the threads in parallel. The threads perform various scheduling schemes according to their thread IDs. The first thread in each group is a manager (Line 8). In addition, the first thread in Group$_0$ i.e. Thread 0 performs as the supermanager (Line 10). The rest of the threads are workers (Line 13). Given thread ID i, the corresponding group is $\lfloor i/Q \rfloor$.

Algorithm 3 shows the meta-scheduling method for the supermanager. The algorithm consists of two parts: updating *rank* (Lines 1-2) and re-grouping (Lines 3-11). We use a heuristic to update *rank*: Note that WI_j is the computational workload for Group$_j$. A large WI_j requires more workers for task execution. $|LCL_j|$ is the number of completed tasks and d is the average number of successive tasks. For each completed task, the manager reduces the dependency degree of the successive tasks and moves ready-to-execute tasks to LRL_j. Thus, $(|LCL_j| \cdot d)$ represents the workload for the scheduler. A larger $(|LCL_j| \cdot d)$ requires more managers for task scheduling. In Line 1, the ratio r tells us if we need more managers or more workers. If more workers are needed, we increase *rank* in Line 2. In this case, groups are merged to provide more workers per manager. Otherwise, *rank* decreases. Line 2 also ensures that *rank* is valid by checking the boundary values. d, δ_+ and δ_- are given as inputs. The re-grouping depends on the value of *rank*. If *rank* increases, two groups merge (Line 5); if *rank* decreases, the merged group is partitioned (Line 9). The two operators Merge$(\cdot)$ and Partition$(\cdot)$ are discussed in Section 3.2. Line 12 flips f_{exit} if no task remains in GL. This notifies all of the threads to terminate (Line 5 in Algorithm 3).

Algorithm 4 shows an iteration of the group level scheduling for managers. Each iteration consists of three parts: updating WI_i (Lines 1-2 and 15), maintaining precedence relationship (Lines 3-8) and allocating tasks (Lines 9-14). Lines 3-8 check the successors of all tasks in LCL_i in batch mode to reduce

Algorithm 2. A Sample Implementation of Hierarchical Scheduler

Input: P threads; Task dependency graph stored in GL; Thresholds δ_M, δ_+ and δ_-.
Output: Assign each task to a worker thread

{`Initialization`}
1: Group$_j$={*Manager*: Thread$_{2j}$, *Worker*: Thread$_{2j+1}$}, $j = 0, 1, \cdots, P/2-1$
2: Evenly distribute tasks $\{T_i | T_i \in GL \text{ and } d_i = 0\}$ across LRL_j, $WI_j = \sum_{T \in LRL_j} w_T$, $\forall j = 0, 1, \cdots, P/2-1$
3: f_{exit} =**false**, $rank = 1$
{`Scheduling`}
4: **for** Thread $i = 0, 1, \cdots, P-1$ **pardo**
5: **while** f_{exit} =**false do**
6: $Q = 2^{rank}$
7: **if** $i\%Q = 0$ **then**
{`Manager thread`}
8: Group level scheduling at Group$_{\lfloor i/Q \rfloor}$ (*Algorithm* 4)
9: **if** $i = 0$ **then**
{`Supermanager thread`}
10: Meta-level scheduling (*Algorithm* 3)
11: **end if**
12: **else**
{`Worker thread`}
13: Within-group level scheduling at Group $_{\lfloor i/Q \rfloor}$ (*Algorithm* 5)
14: **end if**
15: **end while**
16: **end for**
17: **if** $GL = \emptyset$ **then** f_{exit} =**true**

synchronization overhead. Let $m = 2^{rank} - 1$ denote the number of workers per group. In the batch task allocation part (Lines 9-14), we first fetch m tasks from GRL. Line 12 is an adaptive step of this algorithm. If the overall workload of the m tasks is too light ($\sum_{T \in S'} w_T < \Delta W$) or the current tasks in LRL_i is not enough to keep the workers busy ($WI_i < \delta_M$), more tasks are fetched for the next iteration. This dynamically adjusts the workload distribution and prevents possible starvation for any groups. In Line 10, the manager inspects a set of tasks and selects m tasks with relatively more successors. This is a widely used heuristic for scheduling [2]. Several statements in Algorithm 4 are put into boxes, where the managers access shared components across the groups. Synchronization cost of these statements varies as the number of groups changes.

The workers schedule tasks assigned by their manager (Algorithm 5). This algorithm is a straightforward self-scheduling, where each idle worker fetches a task from LRL_i and then puts the tasks to LCL_i after execution. Although LRL_i and LCL_i are shared by the manager and worker threads in the same group, no worker accesses any variables shared between groups.

Algorithm 3. Meta-Level Scheduling for Supermanager

{Update rank}
1: $r = \sum_{j=0}^{P/Q}(WI_j/(|LCL_j| \cdot d))$, $rank_{old} = rank$
2: $rank = \begin{cases} \min(rank+1, \log P), & r > \delta_+ \\ \max(rank-1, 1), & r < \delta_- \end{cases}$
{regrouping}
3: **if** $rank_{old} < rank$ **then**
{Combine Groups}
4: **for** $j = 0$ to $P/(2 \cdot Q) - 1$ **do**
5: $\text{Group}_j = \text{Merge}(\text{Group}_{2j}, \text{Group}_{2j+1})$
6: **end for**
7: **else if** $rank_{old} > rank$ **then**
{Partition Group}
8: **for** $j = P/Q - 1$ downto 0 **do**
9: $(\text{Group}_{2j}, \text{Group}_{2j+1}) = \text{Partition}(\text{Group}_j)$
10: **end for**
11: **end if**

Algorithm 4. Group Level Scheduling for the Manager of Group_i

{Update workload indicator}
1: $\Delta W = \sum_{\tilde{T} \in LCL_i} w_{\tilde{T}}$
2: $WI_i = WI_i - \Delta W$
{Update precedence relations}
3: **for all** $T \in$ {successors of $\tilde{T}$, $\forall \tilde{T} \in LCL_i$} **do**
4: $d_T = d_T - 1$
5: **if** $d_T = 0$ **then**
6: $GRL = GRL \cup \{T\}$; $GL = GL \backslash \{T\}$
7: **end if**
8: **end for**
{Batch task allocation}
9: **if** LRL_i is not full **then**
10: $S' \Leftarrow$ fetch m tasks from GRL, if any
11: **if** $\sum_{T \in S'} w_T < \Delta W$ or $WI_i < \delta_M$ **then**
12: Fetch more tasks from GRL to S', so that $\sum_{T \in S'} w_T \approx \Delta W + \delta_M$
13: **end if**
14: $LRL_i = LRL_i \cup \{S'\}$
15: $WI_i = WI_i + \sum_{T \in S'} w_T$
16: **end if**

4 Experiments

4.1 Computing Facilities

The Sun UltraSPARC T2 (Niagara 2) platform was a Sunfire T2000 server with a Sun UltraSPARC T2 multithreading processor [4]. UltraSPARC T2 has 8 hardware multithreading cores, each running at 1.4 GHz. In addition, each core supports up to 8 hardware threads with 2 shared pipelines. Thus, there are 64 hardware threads. Each core has its own L1 cache shared by the threads within a core. The L2 cache size is 4 MB, shared by all hardware threads. The platform had 32 GB DDR2 memory shared by all the cores. The operating system was Sun Solaris 11 and we used Sun Studio CC with Level 4 optimization (-xO4) to compile the code.

4.2 Baseline

To compare the performance of the proposed method, we performed DAG structured computations using Charm++ [17] Cilk [14] and OpenMP [16]. In addition, we implemented *three* typical schedulers called `Cent ded`, `Dist shared` and `Steal`, respectively. We evaluated the baseline methods along with the proposed scheduler using the same input task dependency graphs.

Algorithm 5. Within-Group Level Scheduling for a Worker of Group$_i$

Input:
Output:
1: Fetch T from LRL_i
2: **if** $T \neq \emptyset$ **then**
3: Execute task T
4: $LCL_i = LCL_i \cup \{T\}$
5: **end if**

(a) Scheduling DAG structured computations using Charm++ (`Charm++`): Charm++ runtime system employs a phase-based dynamic load balancing scheme facilitated by virtualization, where the computation is monitored for load imbalance and computation objects (tasks) are migrated between phases by message passing to restore balance. Given a task dependency graph, each task is packaged as an object called *chore*. Initially, all tasks with dependency degree equal to 0 are submitted to the runtime system. When a task completes, it reduces the dependency degree of the successors. Any successors with reduced dependency degree equal to 0 are submitted to the runtime system for scheduling.

(b) Scheduling DAG structured computations using Cilk (`Cilk`): This baseline scheduler performed work stealing based scheduling using the Cilk runtime system. Unlike the proposed scheduling methods where we bound a thread to a core of a multicore processor and allocated tasks to the threads, we dynamically created a thread for each ready-to-execute task and then let the Cilk runtime system schedule the threads onto cores. Although Cilk can generate a DAG dynamically, we used a given task dependency graph stored in a *shared* global list for the sake of fair comparison. Once a task completed, the corresponding thread reduced the dependency degree of the successors of the task and created new threads for the successors with dependency degree equal to 0. We used spinlocks for the dependency degrees to prevent concurrent write.

(c) Scheduling DAG structured computation using OpenMP (`OpenMP`): This baseline initially inserted all tasks with dependency degree equal to 0 into a ready queue. Then, using the OpenMP pragma directives, we created threads to execute these tasks in parallel. During executing the tasks in the ready queue, we inserted new ready-to-execute tasks into another ready queue for parallel execution in the next iteration. Note that the number of tasks in the ready queue can be much greater than the number of cores. We let the OpenMP runtime system to dynamically schedule tasks to underutilized cores.

(d) Centralized scheduling with dedicated core (`Cent ded`): This scheduling method bound each thread to a separate core. One thread was the manager and the rest were workers. The input DAG was *local* to the manager. Each worker had a ready task list *shared* with the scheduler thread. There was also a completed task list *shared* by all the threads. The manager was also in charge of all the activities related to scheduling and the workers executed assigned tasks only. Pthread mutex locks were used for the ready task lists and completed task list.

(e) Distributed scheduling with shared ready task list (`Dist shared`): In this method, we distributed the scheduling activities across the threads. This method had a *shared* global task list and a *shared* ready task list. Each thread had a *local* completed task list. The schedulers integrated into each thread fetched ready-to-execute tasks from the global task list, and inserted the tasks into the shared ready task list. If the ready task list was not empty, each thread fetched tasks from the ready task list for execution. Each thread inserted the IDs of completed tasks into its completed task list. Then, the scheduler in each thread updated the dependency degree of the successors of tasks in the completed task list, and fetched the tasks with dependency degree equal to 0 for allocation. Pthreads mutex locks were used for the global task list and the ready task list.

(f) Task stealing based scheduling with distributed ready task list (`Steal`): Although the above baseline `Cilk` is also a work stealing scheduler, it used the Cilk runtime system to schedule the threads, each corresponding to a task. On the one hand, the Cilk runtime system has various additional optimizations; on the other hand, scheduling the threads onto cores incurs overhead due to context switching. Therefore, for the sake of fair comparison, we implemented the `Stealing` baseline; we distributed the scheduling activities across the threads, each having a *shared* ready task list. The global task list was *shared* by all the threads. If the ready task list of a thread was not empty, the thread fetched a task from it at the top for execution and upon completion updated the dependency degree of the successors of the task. Tasks with dependency degree equal to 0 were placed into the top of its ready task list by the thread. When a thread ran out of tasks to execute, it randomly chose a ready list to steal a task from its bottom, unless all tasks were completed. The data for randomization were generated offline to reduce possible overhead due to random number generator. Pthreads spinlocks were used for the ready task lists and global task list.

4.3 Datasets and Data Layout

We experimented with both synthetic and real datasets to evaluate the performance of the proposed scheduler. For the synthetic datasets, we varied the task dependency graphs so that we can evaluate our scheduling method using task dependency graphs with various graph topologies, sizes, task workload, task types and accuracies in estimating task weights. For the real datasets, we used task dependency graphs for blocked matrix multiplication (BMM), LU and Cholesky decomposition. In addition, we also used the task dependency graph for exact inference, a classic problem in artificial intelligence, where each task consists of data intensive computations between a set of probabilistic distribution tables (also known as *potential tables*) involving both regular and irregular data accesses [20].

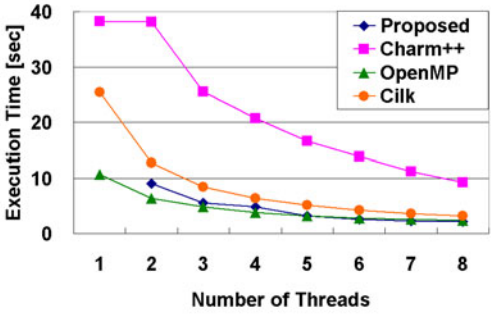
We used the following data layout in the experiments: The task dependency graph was stored as an array in the memory, where each element represents a task with a task ID, weight, number of successors, a pointer to the successor array and a pointer to the task meta data. Thus, each element took 32 Bytes, regardless of what the task consisted of. The task meta data was the data used

for task execution. For LU decomposition, the task meta data is a matrix block; for exact inference, it is a set of potential tables. The lists used by the scheduler, such as GRL, LRLs and LCLs, were circular lists, each having a head and a tail pointer. In case any list was full during scheduling, new elements were inserted on-the-fly.

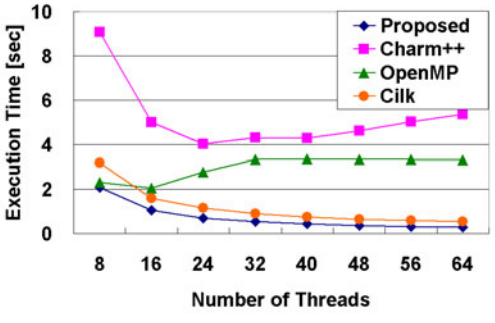
4.4 Results

We compared the performance of the proposed scheduling method with two state-of-the-art parallel programming systems i.e. Charm++[17], Cilk [14] and OpenMP [16]. We used a task dependency graph for which the structure was a random DAG with 10,000 tasks and there was an average of 8 successors for each task. Each task was a dense operation, e.g., multiplication of two 30×30 matrices. For each scheduling method, we varied the number of available threads, so that we could observe the achieved scalability. The results are shown in Figure 5. Similar results were observed for other tasks. Given the number of available threads, we repeated the experiments five times. The results were consistent; the standard deviation of the results were almost within 5% of the execution time. In Figure 5(a), all the methods exhibited scalability, though `Charm++` showed relatively large overhead. A reason for the significant overhead of `Charm++` compared with other methods is that Charm++ runtime system employs message passing based mechanism to migrate tasks for load balancing (see Section 4.2). This increased the amount of data transferring on the system bus. Note that the proposed method required at least two threads to form a group. In Figure 5(b) where more threads were used, our proposed method still showed good scalability; while the performance of the `OpenMP` and `Charm++` degraded significantly. As the number of threads increased, the `Charm+` required frequent message passing based task migration to balance the workload. This stessed the system bus and caused the performance degradation. The performance of `OpenMP` degraded as the number of threads increase, because it can only schedule the tasks in the ready queue (see Section 4.2), which limits the parallelism. `Cilk` showed scalability close to the proposed method, but the execution time was higher.

We compared the proposed scheduling method with three typical schedulers, a centralized scheduler, a distributed scheduler and a task-stealing based scheduler addressed in Section 4.2. We used the same dataset as in the previous experiment, but the matrix sizes were 50×50 (*large*) and 10×10 (*small*) for Figures 6(a) and (b), respectively. We normalized the throughput of each experiment for comparison. We divided the throughput of each experiment by the throughput of the proposed method using 8 threads. The results exhibited *inconsistencies* for the two baseline methods: `Cent ded` achieved much better performance than `Dist shared` with respect to large tasks, but significantly poorer performance with respect to small tasks. Such inconsistencies implied that the impact of the input task dependency graphs on scheduling performance can be significant. An explanation to this observation is that the large tasks required more resources for task execution, but `Dist shared` dedicated many threads to scheduling, which limits the resources for task execution. In addition, many schedulers frequently



(a) Scalability with respect to 1-8 threads

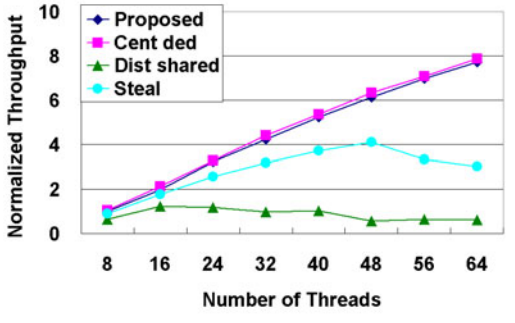


(b) Scalability with respect to 8-64 threads

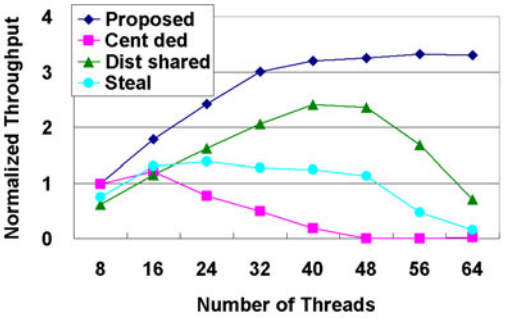
Fig. 5. Comparison of average execution time with existing parallel programming systems

accessing shared data led to significant overheads due to coordination. Thus, the throughput decreased for `Dist shared` as the number of threads increased. When scheduling small tasks, the workers completed the assigned tasks quickly, but the single scheduler of `Cent ded` could not process the completed tasks and allocate new tasks to all the workers in time. Therefore, `Dist shared` achieved higher throughput than `Cent ded` in this case. When scheduling large tasks, the proposed method dynamically merged all the groups and therefore became the same as `Cent ded` (Figure 6(a)). When scheduling small tasks, the proposed scheduler became a distributed scheduler by keeping each core (8 threads) as a group. Compared with `Dist shared`, 8 threads per group led to the best throughput (Figure 6(b)). `Steal` exhibited increasing throughput with respect to the number of threads for large tasks. However, the performance tapered off when more than 48 threads were used. One reason for this observation is that, as the number of thread increases, the chance of stealing tasks also increases. Since a thread must access shared variables when stealing tasks, the coordination overhead increased accordingly. For small tasks, `Steal` showed limited performance compared with the proposed method. As the number of threads increases, the

throughput was adversely affected. The proposed method dynamically changed the group size and merged all the groups for the large tasks. Thus, the proposed method becomes `Cent ded` except for the overhead of grouping. The proposed scheduler kept each core (8 threads) as a group when scheduling the small tasks. Thus, the proposed method achieved almost the same performance as `Cent ded` in Figure 6(a) and the best performance in Figure 6(b).



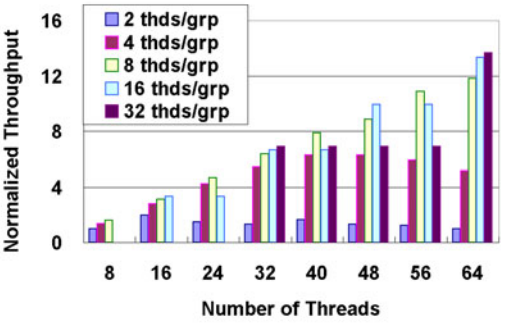
(a) Performance with respect to large tasks (50×50 matrix multiplication for each task)



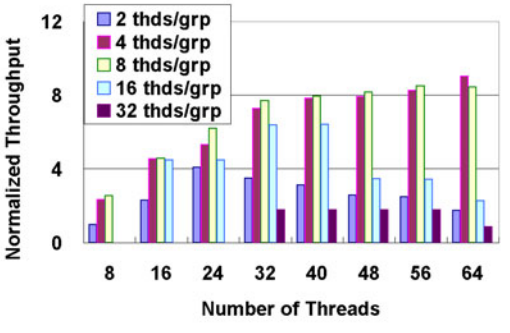
(b) Performance with respect to small tasks (10×10 matrix multiplication for each task)

Fig. 6. Comparison with baseline scheduling methods using task graphs of various task sizes

We experimentally show the importance of adapting the group size to the task dependency graphs in Figure 7. In this experiment, we modified the proposed scheduler by fixing the group size. For each fixed group size, we used the same dataset in the previous experiment and measured the performance as the number of threads increases. According to Figure 7, larger group size led to better performance for large tasks; while for the small tasks, the best performance was achieved when the group size was 4 or 8. Since the optimized group size varied according to the input task dependency graphs, it is necessary to adapt the group size to the input task dependency graph.



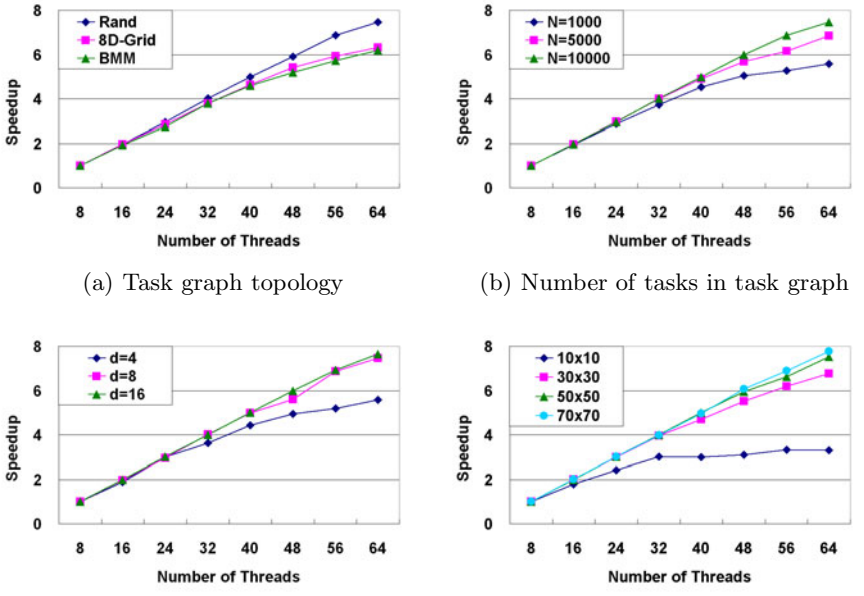
(a) Performance with respect to large tasks (50×50 matrix multiplication for each task)



(b) Performance with respect to small tasks (10×10 matrix multiplication for each task)

Fig. 7. Performance achieved by the proposed method *without* dynamically adjusting the scheduler group size (number of threads per group, *thds/grp*) with respect to task graphs of various task sizes

In Figure 8, we illustrated the impact of various properties of task dependency graphs on the performance of the proposed scheduler. We studied the impact of the topology of the graph structure, the number of tasks in the graph, the number of successors and the size of the tasks. We modified these parameters of the dataset used in the previous experiments. The topologies used in Figure 8(a) included a random graph (Rand), a 8-dimensional grid graph (8D-grid) and the task graph of blocked matrix multiplication (BMM). Note that we only used the topology of the task dependency graph for BMM in this experiment. Each task in the graph was replaced by a matrix multiplication. We evaluate the full BMM as a real-life problem in Figure 13. According to the results, for most of the scenarios, the proposed scheduler achieved almost linear speedup. Note that the speedup for 10×10 task size was relatively lower than others. This was because synchronization in scheduling was relatively large for the task dependency graph with small task sizes. Note that we used the speedup as the metric in Figure 8.



(a) Task graph topology

(b) Number of tasks in task graph

(c) Number of successors of each task

(d) Task size

Fig. 8. Impact of various properties of task dependency graphs on speedup achieved by the proposed method

By speedup, we mean the serial execution time over the parallel execution time, when all the parameters of the task dependency graph are given.

In Figure 9, we investigated the impact of task types on scheduling performance. The computation intensive tasks (Computation) were matrix multiplications, for which the complexity was $O(N^3)$, assuming the matrix size was $N \times N$. In our experiments, we had $N = 50$. The memory access intensive tasks (Mem Access) summed an array of N^2 elements using $O(N^2)$ time. For the last task type (Mixed), we let all the tasks with an even ID perform matrix multiplication and the rest sum an array. We achieved speedup with respect to all task types. The speedup for memory access intensive tasks was relatively lower due to the latency of memory access.

Figure 10 reflects the efficiency of the proposed scheduler. We measured the execution time of each thread to check if the workload was evenly distributed, and normalized the execution time of each thread for the sake of comparison. The underlying graph was a random graph. We also limited the number of available cores in this experiment to observe the load balance in various scenarios. Each core had 8 threads. As the number of cores increased, there was a minor imbalance across the threads. However, the percentage of the imbalanced work was very small compared with the entire execution time.

For real applications, it is generally difficult to estimate the task weights accurately. To study the impact of the error in estimated task weight, we intentionally

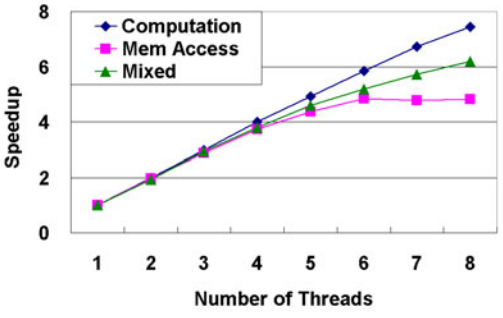


Fig. 9. Performance of the proposed method with respect to computation intensive tasks, memory access intensive tasks and the mix

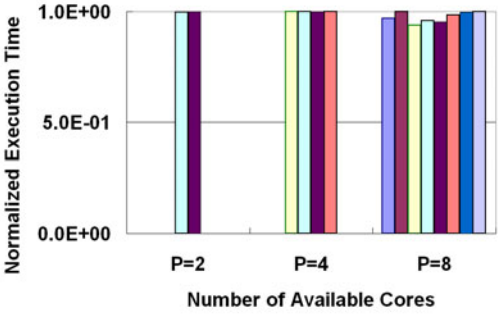


Fig. 10. Load balance achieved by the proposed method with respect to various number of available cores

added noise to the estimated task weight in our experiments. We included noise that added 5%, 10% and 15% of the real task execution time. The noise was drawn from uniform distribution using the POSIX math library. According to the results in Figure 11, the impact was not significant.

In Figure 12, we investigated the overhead of the proposed scheduler. Using the same dataset used in the previous experiment, we first performed hierarchical scheduling and recorded to which thread a task was allocated. According to such allocation information, we performed static scheduling to eliminate the overhead due to the proposed dynamic scheduler. We illustrate the execution time in Figure 12. Unlike the previous experiments, we show the results with respect to execution time to compare both the scalability and the scheduling overhead for a given number of threads. As we can see, the overhead due to dynamic scheduling was very small.

The above experiments were conducted using synthetic datasets, so that we could control the parameters and then study the impact of various factors to the scheduling performance. We achieved consistent results for real application datasets too. In Figure 13, we constructed the task dependency graph according

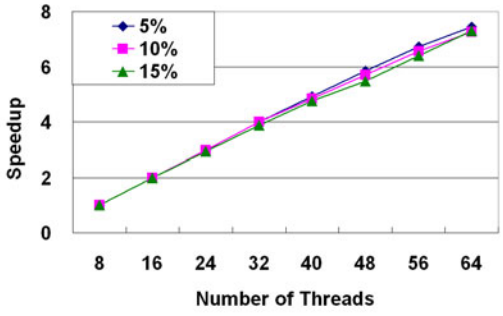


Fig. 11. Impact of the error in estimated task weight on speedup achieved by the proposed method

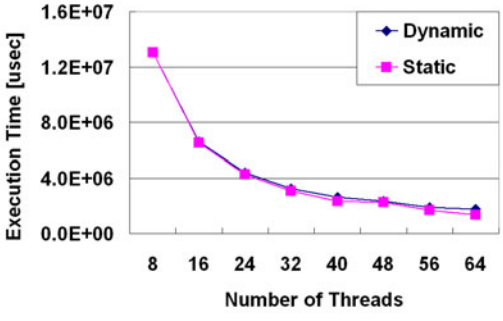


Fig. 12. Overhead of the proposed scheduling method

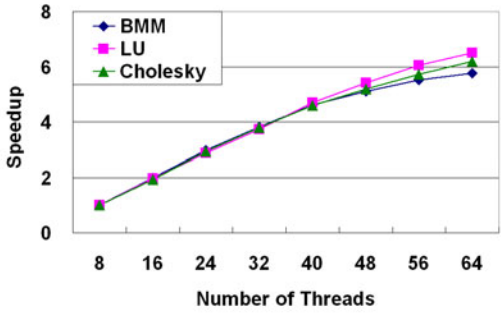


Fig. 13. Performance of the proposed scheduler for real applications

to blocked matrix multiplication (BMM), LU decomposition and Cholesky decomposition [10]. For the BMM, we used a matrix of size 600×600 with block size 50×50. The total number of tasks was 3312. For both the LU and Cholesky decomposition, the matrix size was 1000×1000 and block size was 50×50. The total number of tasks was 2870. In Figure 14, we applied the proposed scheduler for parallel exact inference [20]. The task dependency graph for this problem

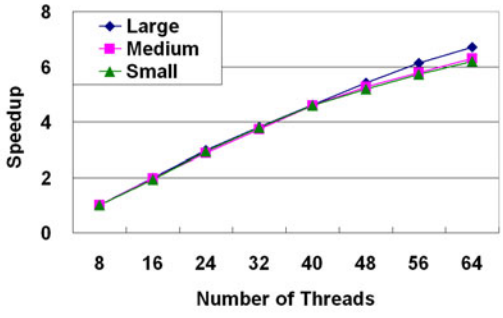


Fig. 14. Performance of the proposed scheduler for exact inference

had 1023 nodes and each node had a potential table of 4096 entries. We manually partitioned the potential tables at different sizes and therefore had three datasets. The sizes of the partitioned potential table were 4096, 1024 and 256 for large, mediate and small tasks, respectively. The proposed scheduler worked well for all the real applications. Note that we used the metric speedup instead of *absolute* execution time or throughput. This is because the absolute performance requires optimization of both the tasks and the scheduler. We only focused on scheduler design in this paper, therefore we used the metric of speedup.

5 Conclusion

We proposed a hierarchical scheduling scheme for manycore processors. In our method, we divided the threads into groups, each having a manager to perform scheduling at the group level and several workers to perform self-scheduling for the tasks assigned by the manager. A supermanager was used to dynamically adjust the group size, so that the scheduler could adapt to the input task dependency graph. We analyzed the proposed method and demonstrated its advantages for manycore architectures. The experimental results on the Sun UltraSPARC T2 processors were encouraging, compared with typical baseline schedulers and existing parallel programming systems. In the future, we plan to study data layout for high throughput processors to efficiently use the data cache of the UltraSPARC processors, since the L2 cache is no more than 4 MB, shared by up to 64 hardware threads. We would also like to explore the heuristics for assigning tasks of various types to a core. For example, interleaving the computationally intensive tasks with memory access intensive tasks may improve the overall performance.

References

1. Ahmad, I., Ranka, S., Khan, S.: Using game theory for scheduling tasks on multicore processors for simultaneous optimization of performance and energy. In: Intl. Sym. on Parallel Dist. Proc., pp. 1–6 (2008)
2. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Computing Surveys 31(4), 406–471 (1999)

3. Zhu, W., Thulasiraman, P., Thulasiram, R.K., Gao, G.R.: Exploring financial applications on many-core-on-a-chip architecture: A first experiment. In: Frontiers of High Performance Computing and Networking, pp. 221–230 (2006)
4. Sheahan, D.: Developing and tuning applications on UltraSPARC T1 chip multithreading systems. Technical report (2007)
5. Tan, G., Sreedhar, V.C., Gao, G.R.: Analysis and performance results of computing betwenness centrality on ibm cyclops64. Journal of Supercomputing (2009)
6. Ahmad, I., Kwok, Y.K., Wu, M.Y.: Analysis, evaluation, and comparison of algorithms for scheduling task graphs on parallel processors. In: Proceedings of the 1996 International Symposium on Parallel Architectures, Algorithms and Networks, pp. 207–213 (1996)
7. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York (1990)
8. Papadimitriou, C., Yannakakis, M.: Towards an architecture-independent analysis of parallel algorithms. In: Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing, pp. 510–513 (1988)
9. Benoit, A., Hakem, M., Robert, Y.: Contention awareness and fault-tolerant scheduling for precedence constrained tasks in heterogeneous systems. Parallel Computing 35(2), 83–108 (2009)
10. Song, F., YarKhan, A., Dongarra, J.: Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In: International Conference for Hight Performance Computing, Networking Storage and Analysis (2009)
11. Coffman, E.G.: Computer and Job-Shop Scheduling Theory. John Wiley and Sons, New York (1976)
12. Karamcheti, V., Chien, A.: A hierarchical load-balancing framework for dynamic multithreaded computations. In: Proceedings of the ACM/IEEE Conference on Supercomputing, pp. 1–17 (1998)
13. Zhao, H., Sakellariou, R.: Scheduling multiple DAGs onto heterogeneous systems. In: IEEE International Symposium on Parallel and Distributed Processing (IPDPS), pp. 1–12 (2006)
14. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. Technical report, Cambridge (1996)
15. Intel Threading Building Blocks, http://www.threadingbuldingblocks.org/
16. OpenMP Application Programming Interface, http://www.openmp.org/
17. Charm++ programming system, http://charm.cs.uiuc.edu/research/charm/
18. Ohara, M., Inoue, H., Sohda, Y., Komatsu, H., Nakatani, T.: Mpi microtask for programming the cell broadband enginetm processor. IBM Systems Journal 45(1), 85–102 (2006)
19. Kurzak, J., Dongarra, J.: Fully dynamic scheduler for numerical computing on multicore processors. Technical report (2009)
20. Xia, Y., Feng, X., Prasanna, V.K.: Parallel evidence propagation on multicore processors. In: The 10th International Conference on Parallel Computing Technologies, pp. 377–391 (2009)
21. Bader, D.: High-performance algorithm engineering for large-scale graph problems and computational biology. In: 4th International Workshop on Efficient and Experimental Algorithms, pp. 16–21 (2005)

Multiplexing Low and High QoS Workloads in Virtual Environments

Sam Verboven, Kurt Vanmechelen, and Jan Broeckhove

University of Antwerp,
Department of Computer Science and Mathematics,
Middelheimlaan 1, 2020 Antwerp, Belgium
sam.verboven@ua.ac.be

Abstract. Virtualization technology has introduced new ways for managing IT infrastructure. The flexible deployment of applications through self-contained virtual machine images has removed the barriers for multiplexing, suspending and migrating applications with their entire execution environment, allowing for a more efficient use of the infrastructure. These developments have given rise to an important challenge regarding the optimal scheduling of virtual machine workloads. In this paper, we specifically address the VM scheduling problem in which workloads that require guaranteed levels of CPU performance are mixed with workloads that do not require such guarantees. We introduce a framework to analyze this scheduling problem and evaluate to what extent such mixed service delivery is beneficial for a provider of virtualized IT infrastructure. Traditionally providers offer IT resources under a guaranteed and fixed performance profile, which can lead to underutilization. The findings of our simulation study show that through proper tuning of a limited set of parameters, the proposed scheduling algorithm allows for a significant increase in utilization without sacrificing on performance dependability.

Keywords: Workload multiplexing, Virtualization, Overbooking, Scheduling.

1 Introduction

A current trend in IT infrastructure management is the reliance on virtualization technology to mitigate the costs of application and IT infrastructure deployment, management and procurement. Virtualization technology allows one to manage an application and its execution environment as a single entity, a *virtual machine* (VM). It allows for the full configuration of an application and its execution environment to be captured in a single file, a virtual machine *image*. These virtual machine images can be deployed in a hardware-agnostic manner on any hardware that hosts a compatible virtual machine *monitor* (VMM) such as Xen [1] or VMware's VMM. VM monitors thereby offer flexibility in partitioning the underlying hardware resources and ensure isolation between the different virtual machines that are running on the same hardware. Aside from the benefits of

E. Frachtenberg and U. Schwiegelshohn (Eds.): JSSPP 2010, LNCS 6253, pp. 175–190, 2010.

this technology in the context of privately owned data centers, these features have also fostered the development of a new IT infrastructure delivery paradigm that is based on outsourcing. The possibility to deploy an entire environment in a low-cost and hardware-neutral manner has paved the way for *cloud* [2,3] infrastructure as a service (IaaS) providers to open up their large datacenters to consumers, thereby exploiting significant economies of scale.

One of the most well known providers in this new market is Amazon with their Elastic Compute Cloud (EC2) [4] offering. As is typical for an IaaS provider, Amazon offers a discrete number of resource types or *instance types* as they are called, with varying performance characteristics. Such an instance type delivers a guaranteed level of compute capacity. An EC2 *small* instance type for example, contractually delivers a performance that is equivalent to a 2007 Opteron processor with a 1.0-1.2 GHz clock frequency. The performance guarantees in this service delivery model are crucial because the use of the compute service is paid for by the hour, and not by actual compute capacity delivered or used. The combination of these performance guarantees and the fact that virtual machine workloads can vary significantly can lead to infrastructure underutilization in absence of corrective measures. In addition, the ability to buy *reserved* instances at EC2 that have a guaranteed level of performance *and* availability, further increases the chances for underutilization. The recent addition of a spot market [5] for EC2 instances whereby instance types are dynamically priced and potentially killed by the provider if their standing bid does not meet the spot price, provides an indication for this problem of (temporary) underutilization. The addition of this market mechanism changes the scheduling problem within the datacenter from one in which a given set of workloads need to be balanced out over the available hardware, to one in which the change of an admission parameter,the instance's spot price, can trigger an influx of additional VM workloads into the datacenter. These workloads run under lower availability guarantees as they can be shutdown by the provider if they cause interference with workloads that run under a high availability regime, such as the reserved instance or on-demand instances at EC2[1].

Scheduling workloads that have low priority and quality of service (QoS) guarantees in terms of performance, alongside with high-QoS workloads thus offers a possibility to deal with underutilization. Consider for example the addition of a batch job workload to a 4-way server that is running a VM with four cores hosting a high priority web service. The web service's spiky load pattern opens up the possibility for filling in underutilized periods with the batch workload. Such a scheduling approach must ensure that high-QoS workloads do not suffer from performance degradation caused by their multiplexing with low-QoS workloads. At the same time, enough low-QoS workloads should pass admission control in order to achieve the highest possible utilization and throughput of the infrastructure.

[1] Note that EC2 uses an indirect mechanism for this by increasing the spot price to a level that rises above the standing bid of an adequately high number of spot instance workloads. This clears them for shutdown under the contractual rules of the trading agreement.

Although some commercial products exist, such as VMware's vSphere, that perform load balancing in a cluster for a given set of virtual machines, no definite solution exists today for tackling this problem if a free decision can be made to accept additional low-QoS workloads. In this paper, we present a simulation framework to analyze the performance of VM scheduling problems and evaluate a scheduling algorithm that is tailored towards the multiplexing of these high- and low-QoS workloads in a virtual machine context. We demonstrate that by tuning a limited set of parameters a tradeoff can be made between maximizing utilization and avoiding workload interference.

2 Model

2.1 Resource and Job Model

In this contribution, we research the VM scheduling problem within the context of the following model. We explore the problem in a setting with one infrastructure provider P, that hosts a set of m machines M_j $(j = 1, \cdots, m)$. These are considered to be identical parallel machines so each machine is able to execute any job from the set of n jobs J_i $(i = 1, \cdots, n)$, and for the machine's processing capacity s_j we have, $\forall i, j \in \{1, \cdots, m\} : s_i = s_j = 1$. A job, which models the execution of a virtual machine instance, has a varying load pattern over time and is sequential, i.e. it runs on only one machine at a time. A job has a release time r_i, and a duration p_i. We consider two types of QoS levels for jobs. High-QoS jobs must be able to start at time r_i and should be able to allocate the full processing power of the machine on which they are deployed. These jobs are not preemptible, e.g. a virtual machine running a relational database. Low-QoS jobs can be preempted at a fixed cost c_p. In this work, we assume that job preemption requires a suspension of the virtual machine. Equivalently, a resumption of a virtual machine instigates a cost c_r. The job startup costs (c_b) and termination costs (c_t) are also modeled as we are dealing with VMs. For preemption, we only consider the case wherein a VM is swapped out of memory to make room for the other VMs that run on the server. An example of a workload that is amenable to a low-QoS regime is a virtual machine that executes low-priority batch jobs.

A machine corresponds to a virtualized core of a server that runs a virtual machine monitor. The provider P operates a cluster of such servers. A machine can accommodate more than one job at a time. We assume that the distribution and multiplexing of a VMs workload over the virtual cores of a server is managed by the virtual machine monitor and do not explicitly model this behavior. We also do not model the overheads that such multiplexing brings in terms of technical considerations such as I/O contention for resources or cache line invalidations. Although these aspects can certainly have a significant impact on this study, they are also very application dependent and difficult to model and simulate. In that respect, this study maps out the maximum performance that can be attained under the proposed scheduling approach.

2.2 VM Management Model and Simulation Framework

For managing the distribution of virtual machines over multiple servers in the cluster a *virtual infrastructure manager* (VIM) is required. There are multiple such managers currently available such as vSphere (VMWare's commercial offering), or one of the open source alternatives such as OpenNebula [6] or Eucalyptus [7]. Depending on the capabilities of the VIM, a set of features and operations is available to manage the execution of the VM instances on the cluster. Because of its generality, we have chosen to model our scheduling problem in the context of the features offered by the OpenNebula toolkit. The open nature of the project, the emphasis on being a research platform and the generality of its feature set are the main factors that influenced this choice.

One of the schedulers already available for OpenNebula is the Haizea [8,6] scheduler. The VM operations available to the scheduler are *shutdown*, *start*, *suspend* and *resume*. The scheduler is assumed to have no knowledge of p_i. In order to deal with infrastructure underutilization, we take an overbooking approach. That is, we allow the scheduler to allocate more resources than physically available on the cluster node. Such an overbooking has to be actively managed by active scheduling decisions in order to limit the interference of low-QoS loads with high-QoS loads. As the Haizea scheduler already supports many of the features required for overbooking, such as the support for differentiation between multiple job types, it is chosen as the basis for our scheduler.

All of the scheduler's decisions result in a series of commands and corresponding VM states that can be used to drive the two enactment backends available in Haizea. The first is a simulated backend used in the presented experiments, the second drives the OpenNebula virtual infrastructure engine where Haizea can be used as an alternative to the default scheduler. One of the major benefits of the second backend is that all the scheduling algorithms implemented within the extended framework are automatically compatible with OpenNebula. An advantage of this choice is that the results of our simulation studies can be verified in a real-world setting without much additional cost.

Haizea's simulation mode uses a simulation core that keeps track of all *actions* that are scheduled with a specific firing and finishing time. The simulation steps through time by subsequently adjusting the simulator's virtual clock to the time of the next action. At each step, the state of the simulated environment is updated and user code can step in to schedule new actions. A single VM operation, such as suspend, can involve one or more actions, depending on the level of detail in the VM management model. For example, one could explicitly model the time required for state checkpointing, or the I/O operation involved in storing the checkpoint.

With a configurable time frequency, our scheduler performs an *overbooking* step. In such a step all available machines are polled to obtain the active jobs and their current utilization. This information is then used to determine all the VM operations that are required, based on the scheduling policy's options. Interspersed with these fixed steps lie *management* steps. During the management steps all events that do not coincide with overbooking step times are performed e.g. issuing a shutdown command when a VM has finished its workload.

3 Scheduling Algorithm

Any overbooking scheme will have the same general goal: reduce resource wastage due to underutilization while at the same time having a minimal impact on the existing resource users. As a result of their suspend and resume capabilities VMs are uniquely suited for this goal provided they have different types of QoS requirements. A lower priority VM can be suspended and resumed at a later, more opportune time and/or location without losing any performed work. The scheduler determines the suitability of machines for low-QoS jobs and only launches the job if sufficient resources are available. For high QoS jobs, the scheduler installs reservations to make sure resources are available for the entire duration of the workload. Low-QoS jobs are queued up until machines are available.

As jobs only use a single machine, the amount of jobs supported by a single cluster node can be expressed in *slots*. Each slot is equivalent to the processing capability of a single CPU core. As such, we will refer to a machine M_i as a slot in the remainder of this paper. Slots provide a convenient abstraction to specify both the available physical resources as well as the maximum allowed amount of overbooking.

High-QoS jobs may require the full processing capacity of the reserved slots at some point in time but it is reasonable to assume this is not permanently the case. The reserved but unused resources pose both an opportunity and a challenge. There is an opportunity to increase overall utilization by scheduling in low-QoS workloads. Depending on the QoS guarantees, interference with high-QoS workloads with must be completely avoided or kept within reasonable bounds. In contrast to the EC2 approach, we want to preserve the work that has been completed in a low-QoS VM and therefore do not kill it if it is detected to interfere with high-QoS VMs. Therefore, our scheduler must take into account the overheads of suspending and resuming low-QoS VMs. Suspending as well as starting and stopping a VM can be a resource intensive operation. Depending on the configuration of the cluster, it is possible that all four major resources (CPU, memory, disk and network) are heavily taxed.

We quantify the interference between VMs by measuring the CPU utilization on a node in excess of 100%. As mentioned before, this is only one dimension of interference that can exist between VMs that are deployed on the same node. Other dimensions such as contention for disk I/O bandwidth will be investigated in future work.

A simple and effective method to put restrictions on the allowed ranges for overbooking is the introductions of bounds. The base algorithm determines its actions using a lower and an upper bound. The lower bound puts a limit on the maximum node utilization for nodes where new low QoS VMs are booted. The upper bound is used to decide when a VM should start suspending. Keeping in mind the overhead of starting and suspending a VM, the algorithm will not schedule more than one of these operations simultaneously one a node.

Our scheduling algorithm works in two steps: scheduling new overbooking requests and evaluating running requests. The first step, for which pseudo-code is shown in Algorithm 1, works as follows. The algorithm starts by obtaining

a list of all the nodes that can currently support an extra VM. The suitability of a node is determined by comparing the node utilization (including the loads introduced by possible overbooked VMs) with a configurable lower bound. All nodes with a utilization lower or equal to this lower bound are added to a list of overbooking candidates. After suitable candidates are found the list of low-QoS requests is updated: incoming requests are added to the back of the queue while suspended requests are added to the front. Suspended requests are ordered by initial arrival time with the oldest appearing at the front of the queue. With all necessary data gathered, VMs can be scheduled until either the available nodes or requests are exhausted.

```
Input: Set of nodes, Set of vm_requests, lower_bound
foreach Node i do
    if Utilization(i) ≤ lower_bound then
        available_nodes.add(i) ;
    end
end
Update(vm_requests) ;
while available_nodes remaining & vm_requests remaining do
    vm = vm_requests.pop() ;
    n = available_nodes.pop() ;
    Schedule(vm on Node n) ;
end
```

Algorithm 1. Adding Overbooked VMs

Since utilization is a volatile property the conditions for overbooking will need to be evaluated at regular intervals. The pseudo code for this part of the algorithm can be found in 2. All nodes supporting one or more overbooked VMs are evaluated, and if the total utilization equals or surpasses the set lower bound the VM that was added last will be suspended.

```
Input: Set of nodes, upper_bound
foreach Node i do
    if Utilization(i) ≥ upper_bound then
        vm = overbooked_vms(i).get_last() ;
        Suspend(vm) ;
    end
end
```

Algorithm 2. Suspending Overbooked VMs

4 Experiments

In this section, we evaluate the performance of our scheduling algorithm. We first outline our experimental setup after which we present and discuss our results.

4.1 Experimental Setup

Our experimental setup consists of three major aspects: the cluster used to deploy the VMs, a list of high- and low-QoS requests and the load generators attached to the requests. The cluster consists of 50 homogeneous octacore nodes. To generate a non-trivial synthetic load pattern that is reminiscent of the behavior of real-world workloads, we introduce the following three different application types[2] following [9]:

Noisy: Starting from a mean utilization value μ, a load pattern is generated by drawing random numbers from a normal distribution $N(\mu, 15)$. An example of a noisy load pattern for $\mu = 75$ can be found in Figure 1.

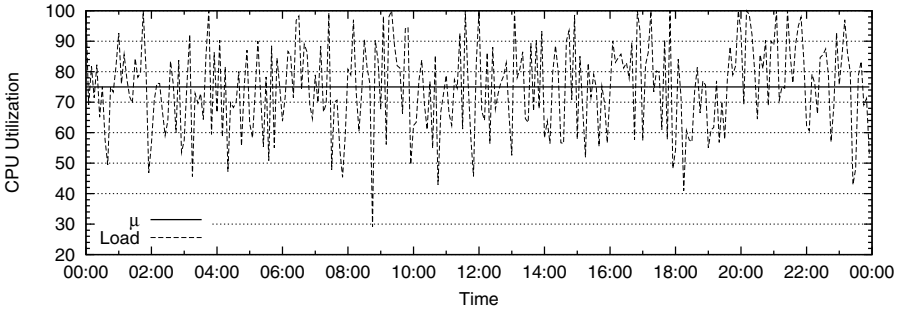


Fig. 1. Sample noisy load pattern

Spiky: This load pattern is based on a normal distribution with $\sigma = 5$. To add load spikes to the pattern, each drawing of the load distribution has 1% chance of generating a spike with 90% chance of having a positive one. Each spike has 50% chance of continuing. An example spiky load pattern for $\mu = 75$ can be found in Figure 2.

Business: A business load pattern is slightly more complicated in that a function is used to determine the μ parameter of the normal distribution $N(\mu, 5)$ depending on the time of day. The value of μ is calculated with a piecewise function that represents utilization fluctuations coinciding with business hours. The function is configured with a minimum (min) and a maximum (max) utilization value. Utilization rises from min to max between 8.00 and 10.00 in the morning. Between 11.30 and 13.30 there is a slight drop representing lunch hours. In the evening there is a second decline dropping back to min between 16.00 and 18.00. The incremental utilization changes between min and max are calculated by adjusting the amplitude and period of a $sinus$ function. During weekends, the function returns the minimum value. An example business load pattern for $min = 50$ and $max = 90$ is shown in Figure 3.

[2] By manipulating a limited number of parameters we can emulate a wide range of applications.

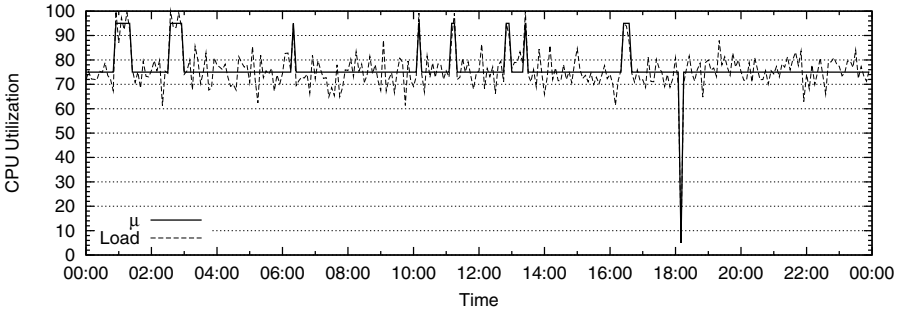


Fig. 2. Sample spiky load pattern

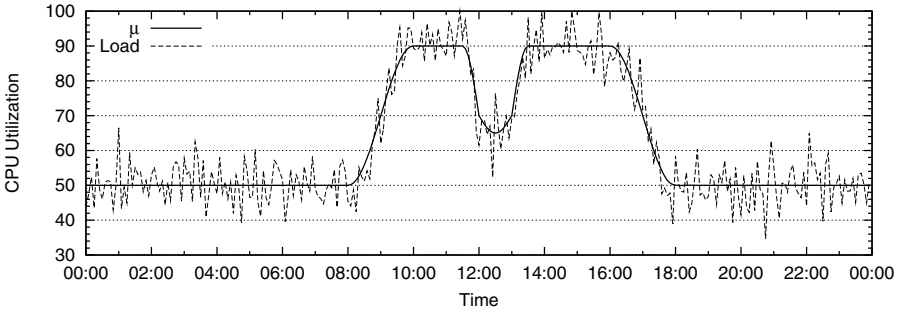


Fig. 3. Sample business load pattern on a weekday

Each high-QoS application has an equal chance of generating one of the three load patterns. For the spiky and noisy load patterns, μ is drawn from a normal distribution $N(75, 15)$. For the business load pattern, $min = 50$ and $max = 90$. An example of a possible workload during a weekday on a node in the cluster can be found in Figure 4. High-QoS applications are generated in such a manner that load patterns are randomly scheduled among the different nodes on the cluster. Each low-QoS application has a noisy load pattern with $\mu = 90$ simulating CPU intensive batch jobs. Each separate application consists of a single job. High-QoS jobs are generated in such a manner that all physical slots are continuously occupied. Using 50 octacore nodes this means there are 400 high-QoS applications running at any given time, one for each core. The low-QoS job arrival rate is set a level that ensures the queue never becomes empty. The maximum amount of concurrently executing low-QoS applications depends on the overbooking slots per node.

All application runtimes are generated according to a geometrical distribution. If X is the runtime in minutes, the probability is expressed in equation 1 for $n = 30, 60, 90, ...$ with p equaling 0.1% and 1% for respectively high- and low-QoS applications.

$$Pr\left[X = n\right] = p\left(1 - p\right)^{\left(\frac{n}{30} - 1\right)} \tag{1}$$

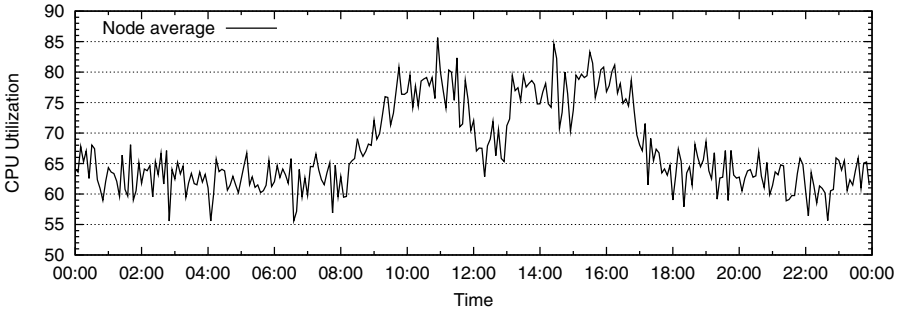


Fig. 4. Sample load pattern on a single eight core node during a weekday

Preliminary tests indicated that the results for running the simulation for one week and for one month produced equivalent results. This is a logical consequence of the weekly repeating pattern. To reduce the time needed to produce results for the numerous tests, we reduced the time horizon of the simulation to one week. The frequency for running the overbooking logic was set to 5 minutes. The costs for VM operations were configured as $c_b = c_p = c_r = c_t = 30s$. Providing an estimate for VM operations in a cluster environment depends highly on not only the storage and network configuration but also on the target VM memory usage. The 30s estimate should be viewed in the context of a cluster using fast network storage to provide the VM images and VM instances using 1 GB of memory. This assumption removes the need to model migration overhead when resuming VMs on different nodes.

Executing the scenario without overbooking logic results in a mean CPU utilization of 69.4% during a total of 67,200 workload hours. Every test consist of three parameters: available overbooking slots, upper- and lower bound. These are chosen in function of the relatively high average utilization on the simulated cluster. The amount of overbooking slots was taken to either be 1, 2 or 3. We varied the upper and lower bounds in increments of 5 between [85, 95] and [60,80] respectively. Since each CPU core can maximally account for a utilization of 12,5%, the minimum difference between lower and upper bound is taken to be 15%. Relaxing this constraint will often result in immediately suspending the VM once it becomes active. All other lower bounds are set 5% apart going down until 60.

4.2 Results

The outcome of the experiments is gathered into Tables 1-3, each containing the results of the test performed for a set amount of overbooking slots. The first column contains upper and lower bounds. The third column shows the average utilization achieved when the overbooking logic is active. The average utilization of 69.4% achieved without overbooking, increases to more than 87% for the scenario with three slots, a lower bound of 80 and upper bound of 95. A conservative bound configuration of 85-60 using a single overbooking slot, leads to

a utilization of 73.7%. The fourth column contains the hours of workload that have executed within overbooked low-QoS VMs. This total does not include any VM operation overhead, only active VMs can contribute to the total. The fifth column shows the amount of VM suspensions. The second to last column contains the amount of *degradation points* if the results are interpreted without any overhead. Degradation points are all overbooking time steps where a total load was recorded that would impact the high-QoS VMs. The last column contains the amount of degradation points taking into account a 5% overhead.

From the preliminary results in Tables 1, 2 and 3 we can reach some initial conclusions with respect to the parameter variations and their results. For the purpose of this discussion we will refer to the difference between upper and lower bounds as the overbooking *window size*. *Negative effects* are considered to be a combination of increased suspensions (and the resulting resumptions) and an increase in the amount of degradation points at both 95 and 100%.

We will first look at the impact caused by the amount of available overbooking slots. The influence of the amount of overbooking slots is lowest in the scenarios with the lowest bound values. This can be attributed to the fact that the average utilization without overbooking is already relatively high, and only allows for a single overbooked VM when bounds are set low. When the bounds are increased more interesting results can be observed. Moving from one overbooking slot to two yields higher utilization levels and often lower negative effects for similar bound values. A single overbooking slot performs only slightly better when the lower bound is set at 60 and total utilization is lowest. Increasing the slot amount to a maximum of three overbooked VMs on the other hand results in similar utilization levels while having the same or more negative effects. A higher maximum increase in utilization can be achieved but there is a substantial increase in the amount of negative effects as well. It seems that in most cases, two overbooking slots is the most appropriate setting for this type of workload.

A detailed side by side comparison of Tables 1-3 shows that although the numbers may vary, the trends that can be detected are similar. There are two trends that deserve some further discussion, namely the effects of *increasing the lower bound* with regard to a fixed upper bound and *increasing the upper bound* with regard to fixed window sizes.

Increasing lower bounds: The first trend is the effect obtained by increasing the lower bound and keeping all other parameters constant. This results in utilization gains that slowly decrease per step. At the same time we find there is an exponential increase in negative effects. This is illustrated in figure 5, the lower bound is increased in steps of 5 from 60 to 80 creating corresponding overbooking windows [35:15]. The results show that although increasing the lower bound will give better utilization gains, these come at an increasingly higher cost. Figures 6 and 7 further show that this effect is present in all window, upper bound combinations.

Increasing upper bounds: Increasing the upper bound under a fixed window size results in a linear increase in utilization (see Figure 6) while suspensions (and degradation points) remain at roughly the same magnitude (see

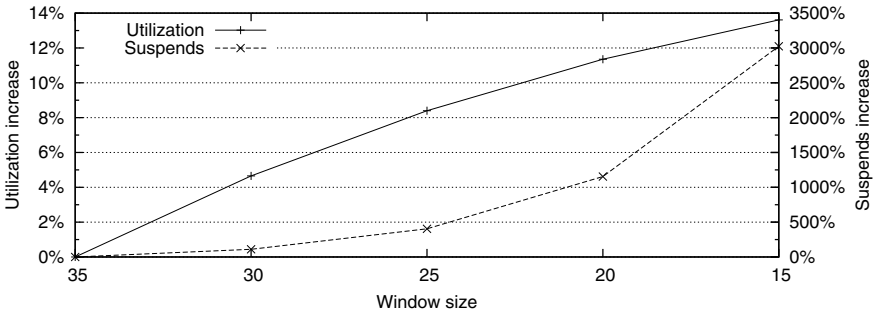


Fig. 5. Increase in utilization and suspensions when using 2 overbooking slots and an upper bound of 95. The lower bound is increased to decrease the overbooking window.

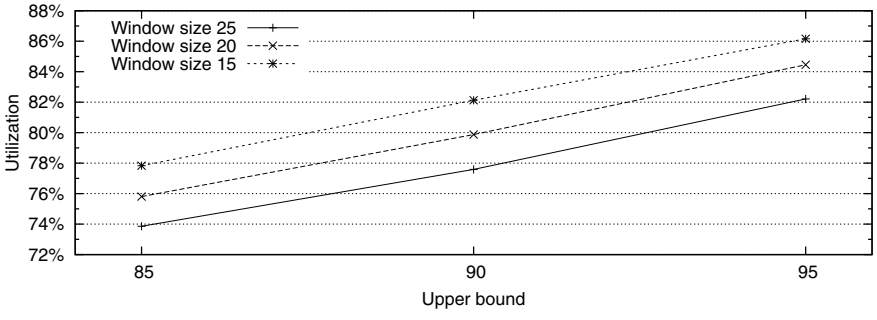


Fig. 6. Utilization with two overbooking slots and varying upper bounds

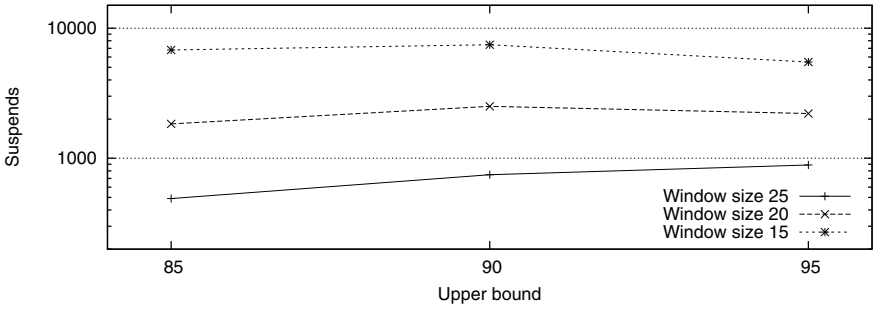


Fig. 7. Suspensions with two overbooking slots and varying upper bounds and windows

Figure 7). From these results we find that choosing a higher upper bound will increase utilization while having a small impact on the negative effects of overbooking.

In summary, we find that selecting a correct amount of *overbooking slots* is an important part of achieving optimal results. There is a tipping point where

Table 1. The results using different bound combinations and one overbooking slot

Bounds	Slots	Utilization	Hours	Suspends	> 100%	> 95%
85 - 60	1	73.7	3297.79	405	0	1
85 - 65	1	75.19	4434.55	1125	0	3
85 - 70	1	76.36	5338.98	3443	0	17
90 - 60	1	74.57	3966.83	191	0	10
90 - 65	1	76.3	5290.09	406	0	25
90 - 70	1	77.42	6150.38	1033	0	62
90 - 75	1	78.28	6806.58	2508	6	195
95 - 60	1	75.25	4491.03	144	3	144
95 - 65	1	77.29	6043.97	234	7	234
95 - 70	1	78.45	6936.92	383	13	384
95 - 75	1	79.17	7478.31	660	22	660
95 - 80	1	79.66	7864.45	1305	64	1305

Table 2. The results using different bound combinations and two overbooking slots

Bounds	Slots	Utilization	Hours	Suspends	> 100%	> 95%
85 - 60	2	73.86	3418.20	490	0	1
85 - 65	2	75.81	4909.43	1839	0	9
85 - 70	2	77.83	6457.53	6797	3	81
90 - 60	2	75.07	4341.34	243	0	15
90 - 65	2	77.59	6277.52	746	1	49
90 - 70	2	79.87	8020.48	2507	7	196
90 - 75	2	82.13	9760.40	7461	49	868
95 - 60	2	75.84	4934.74	176	7	177
95 - 65	2	79.37	7639.71	370	18	371
95 - 70	2	82.21	9813.01	888	43	888
95 - 75	2	84.45	11528.12	2207	135	2208
95 - 80	2	86.16	12845.09	5498	577	5498

extra slots will only add negative effects without additional gain in utilization. We also find that increasing the *lower bound* has diminishing effects on utilization gains while negative effects increase exponentially. On the other hand, increasing the *upper bound* in our current simulator does not add negative effects while utilization displays a steady increase. This leads us to believe that a correct upper bound will most likely depend on limiting factors not yet explored in this research[3]. We can however conclude that the upper bound should be placed as high as possible. Depending on the amount of negative effects an administrator is prepared to allow, an optimal set of bounds can be chosen to maximize utilization.

[3] In multi core systems with more VMs than cores, performance degradation will occur somewhere before total utilization hits 100%.

Table 3. The results using different bound combinations and three overbooking slots

Bounds	Slots	Utilization	Hours	Suspends	> 100%	> 95%
85 - 60	3	73.88	3429.17	476.0	0.0	1.0
85 - 65	3	75.81	4915.34	1866.0	0.0	10.0
85 - 70	3	77.84	6468.15	6871.0	6.0	94.0
90 - 60	3	75.0	4289.79	242.0	0.0	16.0
90 - 65	3	77.57	6266.12	757.0	1.0	51.0
90 - 70	3	79.89	8047.17	2619.0	13.0	211.0
90 - 75	3	82.35	9921.47	8502.0	107.0	1190.0
95 - 60	3	75.88	4964.69	172.0	8.0	173.0
95 - 65	3	79.38	7643.12	376.0	18.0	376.0
95 - 70	3	82.36	9926.95	966.0	53.0	966.0
95 - 75	3	84.92	11885.76	2898.0	246.0	2899.0
95 - 80	3	87.33	13733.74	8838.0	1376.0	8838.0

5 Related Work

To deal with underutilization in batch queuing systems, backfilling techniques such as EASY [10] are often used. Jobs can jump ahead in the queue if they do not delay the start time of the first job in the queue. Conservative backfilling approaches [11] require that upon a backfill operation, no job in the queue is delayed in order to maintain fairness. A problem with these approaches is their reliance on user estimates of job runtimes which are often incorrect [12]. Several techniques have been proposed to model this runtime in order to tackle this problem [13,14,15].

Aside from backfilling, overbooking of resources is another technique to deal with underutilization. The scheduler deliberately overbooks resources in order to deal with jobs that do not use their allocated resource share fully. Sulistio et. al [16] developed a resource overbooking scheme for a setting in which resource reservations are made on a grid infrastructure. Whereas our work hinges on the exploitation of the volatility of VM workloads, their model attempts to deal with the binary case wherein reservations are not used at all or are canceled. They use a richer model for the cost of overbooking by introducing a penalty model that is linked to a renumeration, whereas we only consider the number of performance degradation points the schedule generates. In future work, we are interested in including such an application-specific penalty model to diversify the loss of value an application faces if it is subject to a degradation in performance.

An approach to overbooking non-preemptive workloads in a non-virtualized setting was proposed by Urgaonkar et al. [17]. They demonstrated that controlled overbooking can dramatically increase utilization on shared platforms. Resource requirements are based on detailed application profiling combined with guarantees requested by application providers. The profiling process requires all applications to run on a set of isolated nodes while being subjected to a realistic workload, this workload generates a set of parameters that must be representative for the entire application lifetime. Instead of pro-actively managing

overbooking, application placement is based on a set of constraints and a probability with which these constraints may be violated.

Perhaps somewhat surprisingly, workload traces from the LCG-2 infrastructure, which supports the data processing of CERN's Large Hadron Collider, have shown that as much as 70% of the jobs run by a Tier-2 Resource center in Russia use less than 14% of CPU-time during their lifetime [18]. On the other hand, 98% of the jobs use less than 512MB of RAM. Cherkasova et al. thus investigate the potential of running the batch workloads in VMs and overbooking grid resources to increase utilization. The authors conclude that the use of virtualization and multiplexing multiple VMs on a single CPU core allows for a 50% reduction in the required infrastructure while rejecting less than 1% of the jobs due to resource shortage.

Birkenheuer et al. [19] tackle underutilization for queue-based job scheduling by modeling the probability that a backfill operation in the job queue delays the execution of the next job due to bad user runtime estimations or resource failure. A threshold is defined on this probability to decide whether a job can be used for backfilling. Birkenheuer et al. report on a 20% increase in utilization on a schedule for a workload trace of a 400 processor cluster. Their work is however not adopted to the specifics of virtual machine scheduling and only considers a single-processor case.

At the level of the VMM, priorities and weights can also be assigned to VMs such that high priority workloads maintain their resource share in the presence of low priority loads [20]. The VMM scheduler operates in time quanta that are in the order of tens of milliseconds to ensure the system allocates resources under the configured allocation constraints. Our approach differs from this in that we suspend virtual machines so that their memory pages can be reclaimed by other VMs. Although memory overcommitment is possible in popular VMMs such as Xen, HyperV and VMware, this can result in noticeable performance degradation if the VMs actually require the overcommitted memory [21,22].

6 Future Work

Our first direction of future work will be to further evaluate the effectiveness of the presented scheduling approach. To obtain a complete view a larger amount of slot, bound and scenario combinations must be evaluated. Likewise, we wish to extend the set of workloads that are analyzed and, if possible, make use of trace data from real workloads. Our second goal is to classify VM workloads into predefined classes so that an optimal scheduling configuration can be chosen automatically. Thirdly, we want to improve the scheduling algorithm itself. In this regard we plan to explore the potential of workload modeling and prediction techniques to attain a more intelligent mapping between a low-QoS workload and the cluster node it is placed on. Finally, we plan to add aspects such as memory and network usage to the model in order to increase the accuracy of our results and to allow for the development of more complete scheduling. Using this more accurate model, we will compare our simulation results to those from

OpenNebula experiments conducted with a real backend. In this manner, inconsistencies in the model and its assumptions can be rectified providing a realistic basis for further research.

7 Conclusion

We have introduced a scheduling algorithm which multiplexes low- and high-QoS workloads on a virtualized cluster infrastructure in order to increase the infrastructure's utilization through overbooking. By monitoring the difference between formal and actual requirements of high-QoS workloads in terms of CPU load, an opportunity to add low-QoS workloads to a cluster node is detected. We introduce a limited set of parameters in our scheduling policy so that a flexible tradeoff can be made between maximization of infrastructure utilization and workload interference. The results obtained from initial testing show that depending on the requirements, optimal parameters can be selected that significantly increase utilization while causing limited interference with high-QoS workloads. We identified general trends in the system's performance through parameter tuning and identified a number of guidelines to determine an optimal parameter setting.

References

1. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. SIGOPS Oper. Syst. Rev. 37(5), 164–177 (2003)
2. Weiss, A.: Computing in the clouds. NetWorker 11(4), 16–25 (2007)
3. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Zaharia, M.: Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley (2009)
4. Amazon: Elastic compute cloud, http://aws.amazon.com/ec2 (2008) (accessed 22-12-08)
5. Amazon Web Services LLC: Amazon ec2 spot instances (2009) (accessed December 23, 2009)
6. Sotomayor, B., Montero, R.S., Llorente, I.M., Foster, I.: Virtual infrastructure management in private and hybrid clouds. IEEE Internet Computing 13(5), 14–22 (2009)
7. Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., Zagorodnov, D.: The Eucalyptus open-source cloud-computing system. In: 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID), Washington, DC, USA, pp. 124–131. IEEE, Los Alamitos (2009)
8. Sotomayor, B., Keahey, K., Foster, I.: Combining batch execution and leasing using virtual machines. In: HPDC 2008: Proceedings of the 17th International Symposium on High Performance Distributed Computing, pp. 87–96 (2008)
9. Abrahao, B., Zhang, A.: Characterizing application workloads on cpu utilization for utility computing. Technical Report HPL-2004-157, Hewllet-Packard Labs (2004)
10. Feitelson, D.G., Jette, M.A.: Improved utilization and responsiveness with gang scheduling. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1997 and JSSPP 1997. LNCS, vol. 1291, pp. 238–261. Springer, Heidelberg (1997)

11. Feitelson, D.G., Weil, A.: Utilization and predictability in scheduling the ibm sp2 with backfilling. In: IPPS 1998: Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium, Washington, DC, USA, p. 542. IEEE Computer Society, Los Alamitos (1998)
12. Mualem, A.W., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. IEEE Transactions on Parallel and Distributed Systems 12, 529–543 (2001)
13. Tsafrir, D., Etsion, Y., Feitelson, D.G.: Backfilling using system-generated predictions rather than user runtime estimates. IEEE Trans. Parallel Distrib. Syst. 18(6), 789–803 (2007)
14. Verboven, S., Hellinckx, P., Arickx, F., Broeckhove, J.: Runtime prediction based grid scheduling of parameter sweep jobs. In: APSCC 2008: Proceedings of the 2008 IEEE Asia-Pacific Services Computing Conference, Washington, DC, USA, pp. 33–38. IEEE Computer Society, Los Alamitos (2008)
15. Smith, W., Foster, I.: Using run-time predictions to estimate queue wait times and improve scheduler performance. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 1999, IPPS-WS 1999, and SPDP-WS 1999. LNCS, vol. 1659, pp. 202–219. Springer, Heidelberg (1999)
16. Sulistio, A., Kim, K.H., Buyya, R.: Managing cancellations and no-shows of reservations with overbooking to increase resource revenue. In: CCGRID 2008: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid, Washington, DC, USA, pp. 267–276. IEEE Computer Society, Los Alamitos (2008)
17. Urgaonkar, B., Urgaonkar, B., Shenoy, P., Shenoy, P., Roscoe, T., Roscoe, T.: Resource overbooking and application profiling in shared hosting platforms, pp. 239–254 (2002)
18. Cherkasova, L., Gupta, D., Ryabinkin, E., Kurakin, R., Dobretsov, V., Vahdat, A.: Optimizing grid site manager performance with virtual machines. In: WORLDS 2006: Proceedings of the 3rd conference on USENIX Workshop on Real, Large Distributed Systems, Berkeley, CA, USA, p. 5. USENIX Association (2006)
19. Birkenheuer, G., Brinkmann, A., Karl, H.: The gain of overbooking. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2009. LNCS, vol. 5798, pp. 80–100. Springer, Heidelberg (2009)
20. Cherkasova, L., Gupta, D., Vahdat, A.: Comparison of the three cpu schedulers in Xen. SIGMETRICS Perform. Eval. Rev. 35(2), 42–51 (2007)
21. VMware: Performance best practices for vmware vsphere 4.0 (2009)
22. Microsoft: Virtualization reality: Why microsoft virtualization solutions deliver value when compared to vmware (2009)

Proposal and Evaluation of APIs for Utilizing Inter-Core Time Aggregation Scheduler

Satoshi Yamada and Shigeru Kusakabe

Graduate School of Information Science and Electrical Engineering,
Kyushu University, 744, Motooka, Nishi-ku, Fukuoka, Japan
satoshi@ale.csce.kyushu-u.ac.jp,
kusakabe@ait.kyushu-u.ac.jp

Abstract. This paper proposes and evaluates APIs for Inter-Core Time Aggregation Scheduler (IAS). IAS is a kernel-level thread scheduler to enhance performance of multi-threaded programs on multi-core processors. IAS combines time-multiplexing and space-multiplexing scheduling to utilize caches existing per processing core and shared between processing cores.

We present the effect of APIs in two aspects. Firstly, we show that we can effectively and easily set the aggregation strength in IAS based on the quantum time. Secondly, we show that we can gain the effect of space-multiplexing without setting processor affinity of each thread by grouping processing cores and running IAS per group. We implement IAS and its APIs by modifying a Linux kernel and present its effect on a commodity multi-core processor.

Keywords: Thread Scheduling, Multi-core Processor, Cache Sharing, Multi-threaded Program.

1 Introduction

In this paper, we show the proposal and the evaluation of APIs for Inter-Core Time Aggregation Scheduler (IAS). IAS is a kernel-level thread scheduler to enhance the performance of multi-threaded programs on a commodity multi-core processor. IAS combines time-multiplexing and space-multiplexing scheduler to utilize the caches existing per processing core (Core) and shared between Cores. The contributions of this paper is as follows:

- We show that we can effectively and easily set the aggregation strength in IAS based on the quantum time, which is a period of time that the thread uses CPU.
- We show that we can gain the effect of space-multiplexing without setting the processor affinity of each thread by grouping Cores and running IAS per group.

Nowadays, we have several kinds of multi-core processors, such as Simultaneous Multi-Threading (SMT), Chip Multi-Processing (CMP), and Chip

E. Frachtenberg and U. Schwiegelshohn (Eds.): JSSPP 2010, LNCS 6253, pp. 191–207, 2010.

Multi-Threading (CMT), where we can execute threads in parallel. One of the main differences between multi-core processors and conventional shared-memory multi-processors is that caches, typically L2 caches, are generally shared by Cores in multi-core processors. It is widely known that combinations of threads running simultaneously on different Cores affect the utilization of caches and the performance in a multi-core processor because Cores compete the shared cache with each other[1,2,3]. To utilize the shared cache in a multi-core processor, we propose a thread scheduling mechanism which focuses on multi-threaded programs.

In this paper, a multi-threaded program means a program which executes multiple kernel-level threads sharing the same memory address space in parallel. In Linux, for example, we can implement multi-threaded programs with POSIX library, Java, Perl, MPI, OpenMP, and Open64 because a thread in these languages and compilers systems corresponds to a native thread in the kernel. The rationale of focusing on only multi-threaded programs is that many modern programs, especially commercial programs, are getting multi-threaded as multi-core processors widely spread. For example, database servers and Web servers, such as MySQL and Apache HTTP Server, are multi-threaded to handle multiple client connections efficiently. The modern benchmark programs such as DaCapo Benchmarks[4] and Parsec Benchmark[5] also employ multi-threading to simulate popular and emerging workloads. We expect that we will have more multi-threaded programs and more chances to apply our scheduling mechanism.

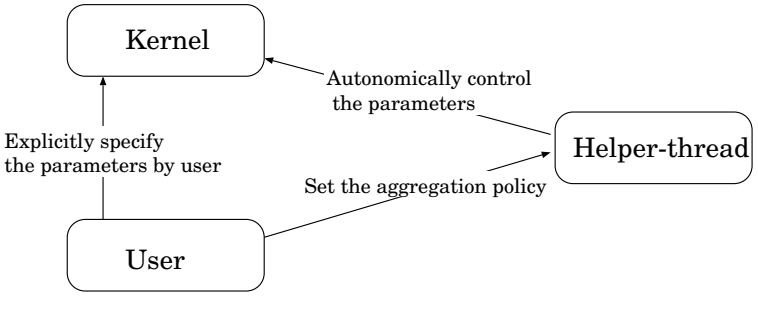


Fig. 1. The overview of the scheduling mechanism. We divide the functions related to scheduling into three domains, which enables the dynamic and flexible control of thread aggregation. In this paper, we focus on User domain.

We show the overview of our scheduling mechanism in Fig. 1. The scheduling mechanism is made up of three domains, Kernel, User, and Helper-thread. Kernel domain provides a basic scheduling mechanism and implemented as IAS. User and Helper-thread are domains which control the parameters for Kernel domain. User domain provides the interfaces to control the parameters explicitly assuming that users are aware of the characteristics of the workloads. Helper-thread domain analyzes the characteristics of the currently executed workloads, detect the degradation of the performance of multi-threaded programs, and controls

the parameters autonomically. Thus, our scheduling mechanism can be applied to the characteristics of the workloads without modifying and re-building Kernel. In this paper, we focus on User domain, and present the APIs to control the behavior of IAS. The detailed design and implementation of Helper-thread domain is our future work.

IAS is a kernel-level thread scheduler for commodity platforms with multi-core processors and implemented by modifying the scheduler of Linux kernel. IAS dynamically aggregates sibling threads, kernel-level threads sharing the same memory address space, and executes them simultaneously on different Cores based on the assumption that sibling threads share a certain amount of working set, the memory area to be accessed by threads. The benefit of IAS is to increase the possibility that co-scheduled threads share their working set and decrease the capacity pressure on the cache. IAS may increase the simultaneous access to the working set, where only transactional access is permitted with locks and semaphores, and cause frequent stalls. However, according to the researches on the analysis of the performance of CMP, the L2 cache misses caused by the insufficiency of capacity are the most influential[3,6]. Therefore, we expect the enhancement of the performance by IAS.

Previously, we investigated the effect of IAS with several multi-threaded benchmark programs and clarified two problems for the effective use of IAS[7,8]. The first problem is the aggregation strength. The effect of IAS depends on the characteristic of programs and platforms such as the size of shared working set between sibling threads of the programs and that of the shared cache size of the platforms. In case sibling threads share a working set, strong aggregations of sibling threads are likely to enhance the performance. On the other hand, IAS can degrade the performance when the workload is I/O intensive and the aggregation of sibling threads results in poor utilization of CPU. For this reason, we should control if we aggregate sibling threads of a program or not, and the aggregation strength. The second problem is the groups of Cores to execute IAS. IAS aggregates sibling threads on the group of Cores specified in the kernel. Previously, we have evaluated the effect of IAS on a dual-core processor. In the dual-core processor, we can make only a single group of Cores. Nowadays, the number of Cores has increased and the structure of the memory hierarchy tends to become complex like Intel Core i7. In such platforms, aggregations of sibling threads with a single group of Cores may increase the overhead of communications between Cores because we assume that sibling threads share a certain amount of working set. Setting processor affinities and assigning Cores to every different program like a conventional space-multiplexing may decrease the communication between Cores. However, it is another difficult issue to optimally set the processor affinity of each thread. We consider that setting multiple groups of Cores to run IAS can reduce the overhead of communications between Cores and we should have an interface to control the groups.

In this paper, we propose and evaluate APIs for IAS to settle the problems mentioned above. We show that we can effectively and easily set the aggregation strength in IAS based on the quantum time of the previously executed thread.

We also show that we can gain the effect of space-multiplexing without setting the processor affinity of each thread by splitting Cores into several groups and running IAS per group.

The rest of this paper is organized as follows. Section 2 explains the implementation and preliminary evaluation of IAS. Section 3 explains the proposal of APIs. Section 4 presents the evaluation of the effectiveness of APIs. Section 5 introduces related works and clarifies our research position. We conclude in Section 6.

2 Implementation and Evaluation of Inter-Core Time Aggregation Scheduler (IAS)

In this Section, we explain the implementation and the evaluation of IAS. We implement IAS by modifying Completely Fair Scheduler (CFS) in Linux 2.6.24 because we assume the use of IAS on commodity processors. IAS ignores the inversion of the priority of each thread in SCHED_NORMAL class, which is non-real-time thread in Linux, and dynamically aggregates sibling threads. We explain the scheduling mechanism of CFS for threads of SCHED_NORMAL class in Section 2.1 and IAS in 2.2. In Section 2.3, we show the preliminary evaluation of IAS on a commodity processor. Based on the preliminary evaluation, we show the problems of running IAS and necessity of effective APIs.

2.1 Completely Fair Scheduler (CFS)

CFS is the standard thread scheduler employed in Linux since its version 2.6.23. CFS is designed to equally distribute CPU time to threads with the same static priority. CFS counts the quantum time of each thread in nanoseconds and calculates the priority as *vruntime* based on the quantum time and `nice` value. When a thread is dispatched by the scheduler, the additional *vruntime* value is calculated from the quantum time and added to the current *vruntime* of the thread. CFS sets higher priority for threads with less *vruntime* to accomplish the fair usage of CPU between threads which start at the same time with the same `nice` value. The runqueues and independent schedulers exist per Core. The load balancer in CFS equalizes the sum of `weight`, which is a value corresponding to `nice` value and defined in the kernel, between runqueues. CFS does not recognize the memory address space of each thread both in scheduling and load balancing.

2.2 Overview of Inter-Core Time Aggregation Scheduler (IAS)

IAS implements two scheduling policies at the same time. The first scheduling policy is the time aggregation, which executes sibling threads in a row on a single Core. The second scheduling policy is the inter-core aggregation, which simultaneously executes sibling threads on different Cores. In this section, we firstly explain Time Aggregation Scheduler (TAS), which is the implementation of the time aggregation. Then, we explain the extension of TAS to adopt the inter-core aggregation.

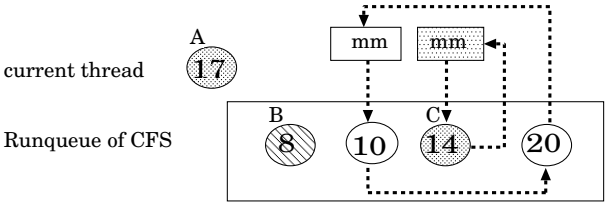


Fig. 2. Example case of TAS. A circle represents a thread and the pattern inside the circle expresses its memory address space. TAS looks for the sibling thread of the current thread from the list of the sibling threads. If there exists a sibling thread (thread C in this case), TAS considers the thread as the candidate for the next thread.

Implementation of Time Aggregation Scheduler (TAS). The basic idea of the implementation of TAS is to dynamically give a priority bonus to the sibling thread of the currently executed thread. As we mentioned in Section 2.1, the priority of a thread is higher when *vruntime* of the thread is smaller. Therefore, the priority bonus for TAS works to reduce *vruntime* of the sibling thread. To implement this idea in CFS, we add a flag to `task_struct`, the structure to maintain the states of a thread in Linux, to recognize if the thread has the sibling threads or not. When a thread creates its sibling thread, TAS sets the flag in `task_struct` and inserts the thread into the list of its sibling threads. The list of the sibling threads exists per Core and sorted in the ascending order of *vruntime*. We show an example case of the time aggregation in Fig. 2.

Fig. 2 shows the runqueue of CFS[1] and the additional links of sibling threads for the time aggregation. The circles in Fig. 2 represent threads and the rectangle containing threads represents a runqueue. The number in a thread shows *vruntime* of the thread. Each thread in Fig. 2 owns *vruntime* of around 10 to 20 for ease of explanation, however, it is common for threads in Linux to own *vruntime* in the millions and the billions calculated from their quantum time counted in nano seconds. Threads are queued in the ascending order of *vruntime* and shown from the left in the runqueue in Fig. 2. The patterns inside the threads represent the memory address spaces. Our scheduler links the sibling threads in the ascending order of their *vruntime*. The links between sibling threads are dashed lines in Fig. 2. We add a member, `mm_sibling`, to the structure of the memory address space, `mm_struct`, in Linux. The links of the sibling threads begin with `mm_sibling` (represented as `mm` in Fig. 2). The currently executed thread A has been dequeued from the runqueue. After executing thread A, CFS selects thread B as the next thread. TAS checks if the flag for the sibling threads is set in thread A. TAS recognizes that the flag is set and looks for the sibling thread from the list of the sibling thread starting from the `mm_sibling` of thread A. TAS finds thread C from the list and considers thread C as another candidate.

[1] The runqueue of CFS has a structure of Red-black tree. We express the runqueue as a list for ease of explanation.

We set the priority bonus for aggregating sibling threads in advance and our scheduler evaluates the expression below.

$$B.vruntime > C.vruntime - priority\_bonus \quad (1)$$

In this paper, we express *vruntime* of a thread as *thread_ID.vruntime* like *B.vruntime*. If expression (1) is true, then TAS will select thread C. If we set the priority bonus equal to or larger than 7 in Fig. 2, TAS will select thread C as the next thread. Otherwise, TAS selects thread B. Thus, TAS is able to aggregate sibling threads while considering the priority of each thread. Also, the scheduling algorithm of TAS is $O(1)$ because the link of sibling threads is sorted in ascending order of *vruntime*.

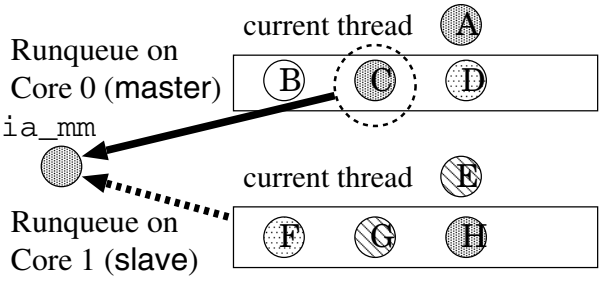


Fig. 3. Example case of IAS. When sibling threads (circles with the same pattern) are aggregated in Core 0 by TAS, the memory address space of the sibling thread is registered in `ia_mm`. The scheduler on Core 1 recognizes for threads sharing the same memory address space by looking at `ia_mm` and considers the thread as the candidate for the next thread.

Extension of TAS to add the inter-core aggregation. We extend TAS to adopt the inter-core aggregation to implement IAS. First of all, we run independent TAS per Core and assign each Core a role of `master` or `slave` Core. IAS lets every Core cooperatively aggregate sibling threads by making `slave` Cores follow the aggregation on `master` Core. When the scheduler on `master` Core finds a chance of aggregating sibling threads, it sets a pointer, `ia_mm`, to the memory address space of the currently executed thread. Otherwise, `ia_mm` is NULL. Only `master` Core can manipulate `ia_mm` while `slave` Cores only refer to `ia_mm`. When `ia_mm` is set to an actual memory address space, the schedulers on `slave` Cores look for the sibling threads sharing the memory address space, which `ia_mm` points to, in their own runqueue. If there exists sibling threads, the schedulers consider the threads as the candidates for the next thread to be scheduled with the priority bonus.

We show an example case of IAS on a platform of a dual-core processor in Fig. 3. In Fig. 3, the circles represent threads and squares represent runqueues on each Core. The pattern inside the thread represents the memory address space and three threads are waiting in the runqueue on each Core. While we

omit *vruntime* values in Fig. 3, threads are enqueued into each runqueue in the ascending order of their *vruntime* from the left. Thread A is running on `master` Core and thread E is running on `slave` Core. We also omit the links between sibling threads in Fig.3. Thread B on `master` Core and thread F on `slave` Core are to be scheduled next to thread A and thread E in case of CFS. After executing thread A on `master` Core, thread C is also the candidate to be scheduled next in TAS because thread C is the sibling thread of thread A. If thread C satisfies expression (1), the scheduler on `master` Core sets the memory address space of thread C to `ia_mm` (solid arrow in Fig. 3). On `slave` Core, the scheduler checks `ia_mm` in scheduling (dashed arrow in Fig. 3). After executing thread E, thread F, G, and H are the candidates because thread G is a sibling thread of thread E and thread H is a sibling thread sharing the memory address space set in `ia_mm`. To execute sibling threads simultaneously on different Cores, IAS raises the priority of the thread sharing the memory address space, which `ia_mm` points to, with the priority bonus. Thread H has the priority bonus against thread F and thread G. If thread H satisfies both expression (2) and (3), thread H will be scheduled after thread E.

$$F.vruntime > H.vruntime - priority\_bonus \tag{2}$$

$$G.vruntime > H.vruntime - priority\_bonus \tag{3}$$

Following the steps above, IAS can execute sibling threads nearly simultaneously on different Cores while considering the priority of each thread. When thread H does not satisfy expression (2) and (3), IAS behaves as TAS. If expression (4) is satisfied, thread G will be the next thread. If expression (4) is not satisfied, thread F will be scheduled.

$$F.vruntime > G.vruntime - priority\_bonus \tag{4}$$

IAS uses the link of sibling threads, which we use for the time aggregation, to search for the sibling threads. The scheduling cost of IAS is also $O(1)$ because the sibling threads are sorted in ascending order in the link.

2.3 Preliminary Evaluation of Inter-Core Time Aggregation Scheduler (IAS)

In this section, we show the preliminary evaluation of IAS in terms of its overhead against CFS. We also show the effectiveness of IAS on RUBiS benchmark [9], which is a benchmark program to measure the performance of a Web application server running a multi-threaded HTTP server and a database server simultaneously. Firstly, we show that the overhead of IAS is small compared to CFS. Then, we show that the effect of IAS depends largely on the value of the priority bonuses [8], indicating that an easy and effective way of controlling the priority bonus is necessary.

Overhead of IAS. We evaluate the additional overhead of IAS compared to CFS. The following tasks are causes of the overhead of IAS.

- Setting the flag of sibling threads in the added member of `task_struct`
- Setting link between sibling threads in the runqueues
- Considering sibling threads in scheduling

We implement a benchmark, which measures the execution time of creating and joining multiple sibling threads, to evaluate the total additional overhead of IAS. The created sibling threads just join with the parent thread. Also, we set the priority bonus for IAS as 0 to schedule threads according to the priority of CFS. We compare the execution time in CFS and IAS and measure the sum of the listed overhead.

According to our measurements, we see the increase of the execution time in IAS by 1% in creating and joining 500K sibling threads. In case the aggregation of sibling threads degrades the performance, we only have to set the priority bonus as 0.

Table 1. Result of RUBiS benchmark

Kernel	CFS	IAS		
		1M*vruntime*	10M*vruntime*	100M*vruntime*
Completed Sessions	230	259 (1.12)	301 (1.30)	265 (1.15)
Response Time (ms)	62,556	48,760 (0.77)	43,090 (0.68)	53,230 (0.85)

Effect of IAS on RUBiS benchmark. We show the effect of IAS in running RUBiS benchmark in Table 1. RUBiS is a benchmark application which simulates the workload of ebay.com and evaluates the performance of a Web application consisting of a HTTP and a database server. RUBiS sends simultaneous requests from multiple clients to the Web application server and evaluates the throughput (Completed Sessions) and the average response time (Response Time) of each request. Both HTTP (Apache HTTP server 2.2.8) and database servers (MySQL 5.0.45) are multithreaded, therefore, IAS aggregates threads of both servers. We use RUBiS benchmark because each thread of these transaction-oriented applications is likely to share the working set rather than scientific application benchmarks[10,11]. We change the value of the priority bonus and compare the result with CFS. The numbers in the parentheses indicates the ratio of the result in IAS against CFS.

In Table 1, we see the increase of the throughput and the reduction of the response time in IAS compared to CFS, indicating IAS is effective in enhancing the performance of a Web application server. We also see that the effect varies as we change the priority bonus and we have to set the priority bonus around 10 millions to maximize the effect. When we set the priority bonus as high as 100 millions *vruntime*, IAS aggregates too many sibling threads of one server and let the sibling threads of another server wait too long. The result shows that we have to tune the priority bonus to accomplish the better performance in running multiple multi-threaded programs.

2.4 Problems of IAS

Based on the preliminary evaluation in Section 2.3, we consider two problems in running IAS as shown below.

- Control of the priority bonus
- Allocation of `master/slave` role

Firstly, the effectiveness of IAS depends on the characteristic of each program. In case IAS degrades the total performance by the aggregation of some programs, users should have an interface to set the priority bonus as 0 or tell the kernel not to aggregate the sibling threads of those programs. Even when IAS enhances the total performance by aggregating the sibling threads of some programs, the priority bonus should be given in proper strength to maintain some degree of fairness of CPU usage between threads. Assuming users are aware of the characteristics of each program in advance, it is still difficult to properly give the priority bonus in *vruntime*. As we mention in Section 2.1, *vruntime* is calculated in the order of nano seconds and too fine-grained for users to control the behavior of the scheduler. We consider that users should have an interface to control the aggregation strength other than specifying the priority bonus in *vruntime*.

Secondly, users should have an interface to allocate multiple groups of `master/slave` flexibly. Nowadays, we have multi-Core processors with complex memory hierarchy. For example, Intel Core 2 Quad has four Cores. Each Core has own L1 data/instruction cache and a single L2 cache is shared between two Cores, while no cache is shared between all Cores. In this case, aggregating sibling threads with a single `ia_mm` may increase the overhead of communication between Cores not sharing the same L2 cache. We consider that users should have an interface to allocate multiple groups of `master/slave` Cores.

3 APIs for Inter-Core Time Aggregation Scheduler (IAS)

In this section, we propose the APIs for IAS, `set_ias_agg` and `set_ias_alloc`, which deal with the problems described in Section 2.4. In Section 3.1, we explain `set_ias_agg`, which controls the strength of aggregation of sibling threads. In Section 3.2, we explain `set_ias_alloc`, which controls the assignment of `master/slave`.

3.1 `set_ias_agg`

There are five arguments passed to `set_ias_agg` as shown below.

- *pid*
- *agg*
- *bonus_type*
- *bonus_value*
- *limit*

We specify the process ID to control the aggregation of its sibling threads by *pid*. At the implementation level, `set_ias_agg` sets the values of *agg*, *bonus_type*, *bonus_value*, and *limit* to the members added in the data structure of memory address space of thread *pid*. The values stored in the members in the memory address spaces are the parameters for IAS to make scheduling decisions. We explain each member below.

agg must be 0 or 1. If *agg* is 0, IAS does not aggregate sibling threads of *pid*. If *agg* is 1, IAS aggregates sibling threads of *pid*. The kernel initializes the values of *agg* as 0 and IAS does not aggregate any threads by default. Users should set *agg* as 1 only when they judge that the aggregation of the sibling threads is effective.

IAS provides two ways to specify the priority bonus with *bonus_type* and *bonus_value*. *bonus_type* takes 0 or 1. If *bonus_type* is 0, IAS gives the priority bonus in *vruntime* specified in `bonus_value`. In this case, `bonus_value` ranges from 0 to over 18,446,744,073G *vruntime*[2]. If *bonus_type* is 1, IAS gives the priority bonus by multiplying the quantum time of the previously executed thread by *bonus_value*. There are four reasons to utilize the quantum time of previously executed thread. Firstly, the change of the additional *vruntime* influences the order of threads in the runqueues. We assume that parallel tasks are equally assigned to sibling threads during their execution. In this case, the difference of *vruntime* between sibling threads are less than the quantum time of previously executed thread. For this reason, we consider that setting the quantum time as the criterion of the priority bonus is reasonable. Secondly, the quantum time changes dynamically according to the workload, therefore, it is hard for users to statically guess the effective priority bonus. Thirdly, it is easy to evaluate the quantum time because CFS tracks it for the calculation of *vruntime*. Fourthly, it is easier to make a guideline of using IAS between different programs. As we mention, the range of `bonus_value` is too wide to properly decide the effective priority bonus to enhance the throughput while keeping a certain fairness between different programs. For these reasons, we consider that using the quantum time provides a reasonable way of the abstraction.

Users can also restrict the number of sibling threads successively scheduled per aggregation on a single Core by specifying the value of *limit*. IAS counts the number of sibling threads successively selected on a single Core. When the count exceeds the *limit*, IAS does not give the priority bonus to sibling threads and resets the count.

3.2 `set_ias_alloc`

`set_ias_alloc` assigns `master/slave` roles to each Core. The arguments passed to `set_ias_alloc` are numbers which specify the role of each Core. We assume the use of `set_ias_alloc` from command lines because the allocation of `master/slave` influences the execution of all threads in the system and we need to observe the impact while interactively running programs.

[2] *vruntime* has the type of `unsigned long long` and we assume to use 32 bit kernel here.

Table 2. The correspondence between the number and its role in `set_ias_alloc`

Number in `ias_job_alloc[]`	Correspondent Role
0	master_0
1	slave_0
2	master_1
3	slave_1
4	master_2
5	slave_2
6	master_3
7	slave_3

IAS controls the role of each Core by using an array `ias_job_alloc[]`, which we defined inside the kernel. The index of `ias_job_alloc[]` corresponds to the ID of Core starting with 0. For example, the role of Core 2 is stored in `ias_job_alloc[2]`. So far, IAS is able to deal with octa-core processors and the role of each Core is specified with numbers from 0 to 7. We show the correspondence between the numbers and its role in Table 2. In Table 2, Cores on `slave_0` follow the aggregation of Core on `master_0`. Following command sets two inter-core aggregation groups on a quad-core processor, one inter-core aggregation group consists of Core 0 and 1 and another group Core 2 and 3.

```
$ set_ias_alloc 0 1 2 3
\widehat{}
```

4 Evaluation of APIs with `memory` Program in SysBench

In this section, we evaluate the effectiveness of APIs with `memory` program in SysBench[14]. In Section 4.1, we explain `memory` program, our experimental platform, and the method of the evaluation. In Section 4.2, we explain the result and show that our API is effective in utilizing IAS.

4.1 `memory` Program and Experimental Platform

SysBench benchmark suites is a collection of benchmark programs to evaluate the performance of workloads related to Online Transaction Processing. `memory` program in SysBench focuses on the performance of sequential reads from or writes to a memory block. `memory` program creates sibling threads and lets them repeat accessing a specified size of shared or unique memory block until the total accessed size exceeds a user-specified size. There are several metrics in `memory` program such as the average time of each data access and the total elapsed time. We can control `memory` program through the parameters such as the number of threads, the size of the memory block, and the total access size.

We show the parameters used for the evaluation in Table 3. In the following explanation, we show the used parameters in the parentheses. We execute 10

Table 3. Parameters for evaluating `memory` program

Parameter		Specified value
-num-threads		100
-memory-oper		write
-memory-scope		global
-memory-block-size	`set_ias_agg`	4(MB)
	`set_ias_alloc`	1, 2, 4, 6, 8, 10 12, 14, 16(MB)
-memory-total-size	`set_ias_agg`	10 (GB)
	`set_ias_alloc`	5, 10, 15 20, 25(GB)

Table 4. Specification of our experimental platform

Processor	Intel Core 2 Quad
L2 Cache Size / Latency	3MB×2 / 5.6 ns
Memory Size / Latency	1.8GB / 74.4 ns
OS / kernel	CentOS 5.3 / Linux 2.6.24

`memory` programs simultaneously to mingle threads of different memory address spaces. We let each program create 100 sibling threads (–num-threads=100) and let sibling threads access the shared memory block (-memory-scope=global) to focus on the effect of utilizing the locality between sibling threads. Each thread writes to the memory block sequentially (–memory-oper=write). We can control the size of the memory block (–memory-block-size) and the total access size (–memory-total-size). We use different values for –memory-block-size and –memory-total-size in the evaluation of each API and explain them in the evaluation method below.

We also show our experimental platform in Table 4. Intel Core 2 Quad is quad-core processor and has two L2 caches, each of which is shared by two Cores.

We measure the total elapsed time and the number of resource stalls (RESOURCE_STALLS.ANY [15]), and compare the results in CFS and IAS with APIs. We show the method of the evaluation in each API below.

Evaluation Method of `set_ias_agg`. In the evaluation of `set_ias_agg`, we focus on the function of setting the value of the priority bonus. We use a single value for the –memory-block-size and the –memory-total-size as shown in Table 3 and set the same parameter to ten `memory` programs. We compare the results of the different methods of setting the priority bonus. We directly specify it in *vruntime* or calculate by multiplying the quantum time of the previously executed thread. In case of setting `bonus_type` as 0, we try wide range of `bonus_value` from 1K to 10M *vruntime* because it is difficult to previously guess the effective value. In case of setting `bonus_type` as 1, we multiply the quantum time by 1 to 5.

Evaluation Method of `set_ias_alloc`. In the evaluation of `set_ias_alloc`, we use five different –memory-total-size values for ten `memory` programs as shown in Table 3, assuming a situation when a user executes several different programs. We also change –memory-block-size to investigate the relationship between the size of the shared working set and the effect of IAS in setting multiple `master/slave` groups.

We prepare three cases, where we set different `master/slave` groups and processor affinity of the threads, and compare their results. The first case is to use a single `master/slave` group, where Core 0 is `master_0` and other three Cores are `slave_0`, and not to set the processor affinity to any threads (Case 1). We set two `master/slave` groups, where Core 0 is `master_0` and Core 1 is `slave_0` while Core 2 is `master_1` and Core 3 is `slave_1`, in the second and the third case (Case 2 and Case 3). The difference between Case 2 and Case 3 is the setting of the processor affinity of threads. In Case 2, we do not specify the processor affinity of threads and threads can be executed in every Core. In Case 3, we divide `memory` programs into two groups as programs of the same total size are split into different `master/slave` group. For example, sibling threads of a `memory` program with –memory-total-size of 10GB are executed on Core 0 and Core 1 while sibling threads of another `memory` program with –memory-total-size of 10GB are executed on Core 2 and Core 3. By specifying the processor affinity as described above, we can divide the workload equally into two Core groups with different L2 caches and restrict the overhead of communication between Cores. We expect the optimal performance in Case 3 and evaluate how close the result in Case 1 and 2 will be. We set the priority bonus as 50M *vruntime* based on our previous experiment [7].

4.2 Results

In this section, we firstly show the results of the evaluation of `set_ias_agg`. Succeedingly, we show the results of the evaluation of `set_ias_alloc`.

Results of the evaluation of `set_ias_agg`. We show the result of the evaluation of `set_ias_agg` in Fig. 4. In Fig. 4, we show the ratio of the execution time in IAS against CFS (lines), and the absolute value of the resource stalls (bars) in each parameter. In Fig. 4, we express each parameter as `d_[1,2,3,4,5]` when we set `bonus_type` as 1, and `s_[1K,10K,100K,1M,10M]` when we set `bonus_type` as 0. We see that the reduction of the execution time and the resource stalls becomes larger as we increase the value of the parameter when we set `bonus_type` as 1. On the other hand, we see little effect of IAS when `bonus_value` is from 1K to 100K when we set `bonus_type` as 0. When we set `bonus_value` higher than 1M *vruntime*, we see the effect becomes larger. We consider that `bonus_value` below 1M *vruntime* is too small in this experiment because the average additional *vruntime*, which we measure simultaneously during the experiment, is 33M.

We conclude that we can set the priority bonus easily and effectively by setting the priority bonus based on the quantum time rather than specifying in *vruntime*.

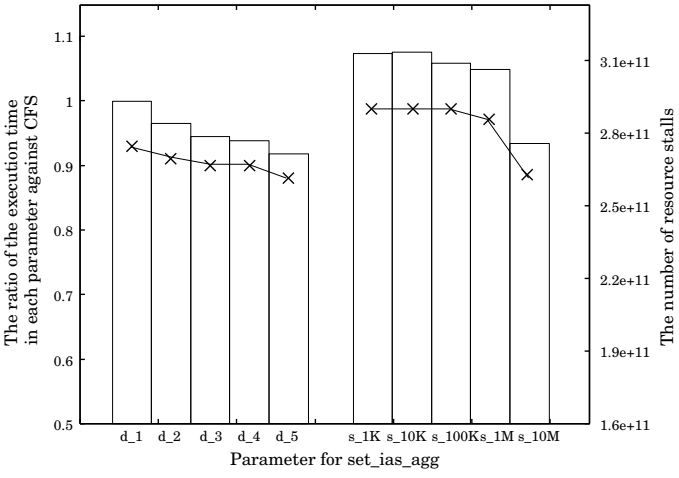


Fig. 4. The effect on the execution time (lines) and the resource stalls (bars) in using `set_ias_agg`

Results of the evaluation of `set_ias_alloc`. We show the result of the evaluation of `set_ias_alloc` in Fig. 5. In Fig. 5, we show the ratio of the execution time in IAS against CFS in Case 1, 2, and 3. We can see the effect in Case 2 and 3 are larger than that in Case 1. We consider that the result shows the effect of space-multiplexing, which reduces the overhead of communication between Cores in Case 2 and 3. We also consider that the effect will be larger in many-core processors with deeper memory hierarchy.

When we compare Case 2 and Case 3, Case 3 seems advantageous only when the memory block is less than 6MB. In other parameters, the effects in Case 2

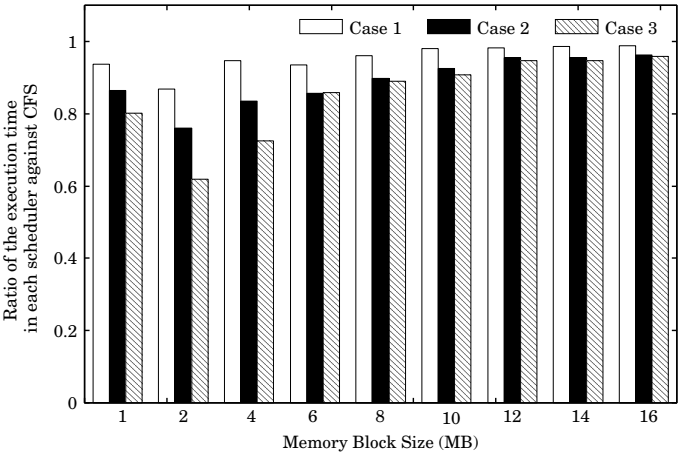


Fig. 5. The effect on the execution time in setting multiple `ia_mm` with `set_ias_agg`

and 3 are almost the same while Case 3 is the optimal setting as we described in Section 4.1. By considering users do not have to set the processor affinity, Case 2 becomes more advantageous as working set gets larger. We conclude that we can gain the effect of reducing the overhead of communication between Cores by setting multiple `master/slave` Cores with `set_ias_alloc`.

5 Related Research

As caches are generally shared between Cores in multi-core processors, many thread-level schedulers have been proposed to utilize the caches. Many researches proposed to split the thread execution into the sampling phase and the scheduling phase [16,17,18]. In the sampling phase, the kernel samples the information of each thread execution. In the scheduling phase, the kernel schedules the combination of threads to execute them simultaneously between different Cores based on the information obtained in the sampling phase. For example, Fedorova [18] calculates the size of the working set of each thread by tracing its behavior in the sampling phase. They schedule the combinations of threads to let the sum of the working set fit within the capacity of the L2 cache. The benefit of this sampling and scheduling approach is that we can apply this method to any case of thread execution in theory. The problem of this approach is the overhead of sampling information, especially when running many threads, as IAS supposes [12,19]. Moreover, the complexity of optimal co-scheduling in multi-core processor, where a cache is shared between all Cores and the number of Cores is more than 2, is NP-complete [2]. We focus on a more realistic approach. Even though IAS does not intend to schedule threads optimally, IAS only focuses on the memory address space of each thread and its overhead is little as we see in Section 2.3.

The basic idea of our approach is similar to that of Chen [3] in that their scheduling algorithm executes threads sharing the working set simultaneously on different Cores to utilize the shared cache. Chen also proposes a compiler to control the granularity of threads to fit with the caches of the processor. Chen's approach is applicable to fine-grained multi-threaded programs, which contains DAGs inside, and shows that their scheduling method can enhance the throughput by carefully tuning the granularity of threads by their compiler. The difference between IAS and Chen's approach is that IAS is intended to work for multi-programmed execution while Chen only considers single-programmed execution. IAS does not detect the size of the working set shared between sibling threads while Chen's approach does not consider the influence from other programs. We consider that we can enhance the performance of broader range of multi-threaded programs by mixing IAS and Chen's approach.

Ziemba also focuses on the locality of references between sibling threads and investigates the effect of space-multiplexing with a Web application server [20]. Ziemba sets different processor affinities for threads of HTTP and application servers in executing SPECweb benchmark [21]. Ziemba presents their aggregation is effective and enhances the performance of the Web application server, indicating the locality of references between sibling threads. However, Ziemba

mentions that it is difficult to statically analyze applications and optimally set processor affinities. In this paper, we present that we can gain the effect of space-multiplexing without setting processor affinities in each thread.

6 Conclusion

This paper proposes and evaluates APIs for IAS, which is a kernel-level thread scheduler to enhance the performance of multi-threaded programs. We have proposed IAS, which dynamically aggregates sibling threads in $O(1)$ to utilize the cache shared between Cores. In this paper, we present two APIs, `set_ias_agg`, which controls the aggregation of sibling threads, and `set_ias_alloc`, which controls `master/slave` groups. The effectiveness of our API is described in two aspects. Firstly, we show that we can effectively and easily set the aggregation strength in IAS based on the quantum time of the previously executed thread by using API `set_ias_agg`. Secondly, we show that we can gain the effect of space-multiplexing by grouping Cores and running IAS per group without setting the processor affinity of each thread by using API `set_ias_alloc`.

Our future work includes the investigation of the effect of IAS with more general benchmark applications. We consider that IAS is especially effective in benchmark applications, which runs multiple multi-threaded programs simultaneously such as SPECweb[21]. We also investigate the effectiveness of Helper-thread mentioned in Section 1. Even though we can set the priority bonus easily with `set_ias_agg`, we still have to set the parameter manually. We will develop Helper-thread mechanism to detect the degradation of multi-threaded programs and automatically tune the priority bonuses to enhance the effect of IAS. In addition, we will develop scheduling strategies to control the behavior of Helper-thread such as the frequency of sampling thread information and the granularity of parameter changes.

References

1. Kim, S., et al.: Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, pp. 111–122 (2004)
2. Jiang, Y., et al.: Analysis and Approximation of Optimal Co-Scheduling on Chip Multiprocessors. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, pp. 220–229 (2008)
3. Chen, S., et al.: Scheduling Threads for Constructive Cache Sharing on CMPs. In: Proceedings of 19th ACM symposium on Parallel Algorithms and Architectures, pp. 105–115 (2007)
4. DaCapo benchmark suite, `http://dacapobench.org/`
5. The PARSEC Benchmark Suite, `http://parsec.cs.princeton.edu/`
6. Chishti, Z., et al.: Optimizing Replication, Communication, and Capacity Allocation in CMPs. In: Proceedings of the 32nd International Symposium on Computer Architecture, pp. 357–368 (2005)

7. Yamada, S., et al.: Development of a Thread Scheduler for Global Aggregation of Sibling Threads. Research Reports on Information Science and Electrical Engineering of Kyushu University 1(2), 69–74 (2008)
8. Yamada, S., et al.: Impact of Priority Bonuses of Inter-Core Aggregation Scheduler on a Commodity CMP Platform. In: Workshop on Managed Many-Core Systems (MMCS) co-located with ASPLOS (2009), http://www.cercs.gatech.edu/mmcs09/program.htm
9. RUBiS: Rice University Bidding System, http://rubis.ow2.org/
10. Keeton, K., et al.: Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads. In: Proceedings of the 25th Annual International Symposium on Computer Architecture, pp. 15–26 (1998)
11. Redstone, J., et al.: An Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture. In: Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 245–256 (2000)
12. DeVuyst, M., et al.: Exploiting Unbalanced Thread Scheduling for Energy and Performance on a CMP of SMT Processors. In: Proceedings of 20th IEEE International Parallel & Distributed Processing Symposium (2006)
13. Yamada, S., et al.: Effect of Context Aware Scheduler on TLB. In: Workshop on Multi-Threaded Architectures and Applications, Published in CD (2008)
14. SysBench: a system performance benchmark, http://sysbench.sourceforge.net/
15. Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3B: System Programming Guide, Part 2, http://www.intel.com/products/processor/manuals/index.htm
16. Parekh, S., et al.: Thread-Sensitive Scheduling for SMT Processors, Technical report, Dept. of Computer Science and Engineering, University of Washington (2000)
17. Snavely, A., et al.: Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor. In: Proceedings of International Conference on Measurement and Modeling of Computer Systems, pp. 66–76 (2002)
18. Fedorova, A., et al.: Performance of Multithreaded Chip Multiprocessors and Implications for Operating System Design. In: Proceedings of USENIX 2005 Annual Technical Conference, pp. 395–398 (2005)
19. Chandra, D., et al.: Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In: Proceedings of 11th International Symposium on High-Performance Computer Architecture, pp. 340–351 (2005)
20. Ziemba, S., et al.: Analyzing the Effectiveness of Multicore Scheduling Using Performance Counters. In: Proceedings of Workshop on the Interaction between Operating Systems and Computer Architecture (2008)
21. SPECweb, http://www.spec.org/web2009/

Using Inaccurate Estimates Accurately

Dan Tsafrir

Department of Computer Science
Technion – Israel Institute of Technology
Haifa 32000, Israel
dants@cs.cs.technion.ac.il

Abstract. Job schedulers improve the system utilization by requiring users to estimate how long their jobs will run and by using this information to better pack (or "backfill") the jobs. But, surprisingly, many studies find that deliberately making estimates *less* accurate boosts (or does not affect) the performance, which helps explain why production systems still exclusively rely on notoriously inaccurate estimates.

We prove these studies wrong by showing that their methodology is erroneous. The studies model an estimate e as being correlated with $r \cdot F$ (where r is the runtime of the associated job, F is some "badness" factor, and larger F values imply increased inaccuracy). We show this model is invalid, because: (1) it conveys too much information to the scheduler; (2) it induces favoritism of short jobs; and (3) it is inherently different than real user inaccuracy, which associates 90% of the jobs with merely 20 estimate values, hindering the scheduler's ability to backfill.

We conclude that researchers must stop using multiples of runtimes as estimates, or else their results would likely be invalid. We develop (and propose to use) a realistic model that preserves the estimates' modality and allows to soundly simulate increased inaccuracy by, e.g., associating more jobs with the maximal runtime allowed (an always-popular estimate, which prevents backfilling).

Keywords: Supercomputing, scheduling, backfilling, user runtime estimates.

1 Context and Background

In a typical supercomputing environment, the supercomputer is a machine that's comprised of up to tens of thousands of nodes, servicing work that is generated by hundreds of users, who collectively submit tens- to hundreds of thousands of jobs. The runtime of jobs ranges from several seconds to tens of hours or more. Jobs can be serial, but more often than not they are parallel. In the context of this paper, "parallel" doesn't mean embarrassingly parallel; rather, each job is comprised of a collection of threads that cooperate and communicate to solve one problem. It is therefore crucial that a job's threads run simultaneously.

Jobs have several attributes, notably, the ID of their submitters (uid), their arrival time, runtime, and size (the number of nodes or processors they require).

E. Frachtenberg and U. Schwiegelshohn (Eds.): JSSPP 2010, LNCS 6253, pp. 208–221, 2010.

Table 1. A typical log file that records the activity of a supercomputer; each line is associated with one submitted job; each column is associated with one job attribute

jobID	*arrival time*	*size*	*runtime*	*estimate*	*uid*	...
1	2010, Apr 24, 12:00:01	2	00:15:37	00:30:00	1013	...
2	2010, Apr 24, 12:05:37	128	01:50:01	18:00:00	1013	...
3	2010, Apr 24, 13:25:20	49	18:00:00	18:00:00	1237	...
...	...	...	...	...	...	...

Table 2. Activity logs we use; see [16] for more details regarding the machines and their workload. We refer to logs in their abbreviated name (leftmost column)

abbrev.	*ver.*	*site*	*cpus*	*jobs*	*duration*	*util.*
CTC	1.1	Cornell Theory Ctr	512	77,222	6/96–5/97	56%
KTH	1.0	Swedish Royal Instit. of Tech.	100	28,490	9/96–8/97	69%
SDSC	2.1	San-Diego Supercomput. Ctr	128	59,725	4/98–4/00	84%
BLUE	2.1	San-Diego Supercomput. Ctr	1,152	243,314	4/00–6/03	76%

We normally think of jobs as rectangles, whereby the vertical dimension is the size, and the horizontal dimension is the runtime. The size if often referred to as the "width" of the jobs (hence jobs can be narrow or wide), and the runtime is often referred to as the "length" of the job (hence jobs can be short or long).

The work submitted by users throughout the lifetime of the machine is recorded in activity logs similar to the one depicted in Table 1.

Many activity logs were collected over the years in the parallel workload archive [16]. Table 2 lists the ones that are used in this study.

2 Backfilling

The baseline scheduling algorithm of most supercomputers is typically rather simple. When a user submits a job, (s)he specifies how many nodes the job needs. The job is then placed in a First-Come First-Served (FCFS) wait queue until enough nodes become free (due to previously submitted jobs that terminate), at which point the job is started, and it runs to completion in batch mode, on a dedicated partition, without ever being preempted. This is illustrated in Figure 1(a).

The problem with FCFS is fragmentation. So all mainstream schedulers employ the optimization that allows job 4 to jump over job 3, provided job 4 doesn't delay job 3, as depicted in Figure 1(b). The act of small jobs jumping ahead before their turn to fill holes in the schedule is called *backfilling* [13].

2.1 Pros of Backfilling: Simple, Effective, and Popular

There are several properties that make backfilling an attractive algorithm: (1) it's simple for users to understand and for developers to implement; (2) it's a batch

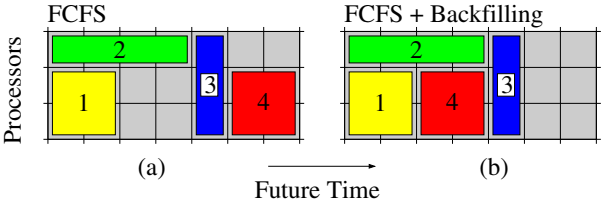


Fig. 1. (**a**) A space/time Gantt chart displaying a FCFS schedule. The X and Y axes denote time and the nodes comprising the machine, respectively. Each rectangle represents a job, such that the rectangle's width and height are the job's runtime and size, respectively. The job numbers indicate arrival order (not arrival time). (**b**) Backfilling reduces fragmentation and improves the utilization by allowing narrow/short jobs to start ahead of their time. Note that it would have been impossible to backfill job 4 had its length been more than 2 time units, as job 3 would have been delayed.

scheduler, which is a virtue in the context of high-performance computing, because applications are often tailored to make use of all available memory, in which case not sharing the memory with others is important; (3) empirical studies show that backfilling improves the utilization of the machines by 10–30 percentage points [12]; and, (4) as it turns out, despite its simplicity, backfilling produces performance results that are a close second to more sophisticated scheduling schemes that, e.g., employ preemption and migration [2,26].

The consequence of the above attractive properties of backfilling is that it became the de-facto standard for supercomputer scheduling. Backfilling is nowadays supported by all the relevant mainstream production products [7], including Load Leveler (by IBM), Maui and Moab (by Cluster Resources), LSF (by platform), OpenPBS and PBS-Pro (by Alair), and GridEngine (by Sun). A survey of the top 50 machines within the top-500-list [4] indicated that 60% of them employ backfilling as their scheduling algorithm [6]. Probably due to its popularity and success, there are many research efforts and papers that deal with backfilling, and many variants were suggested [9].

2.2 Cons of Backfilling: Mandating User Runtime Estimates

There is a price to pay for all the aforesaid attractive properties: in order to operate correctly, a backfilling scheduler must know what's going to happen in the future, namely, it must know in advance how long each job will run.

For example, in Figure 1(b), assume we've just reached T_2 (time unit 2), J_1 (job 1) has just ended, and J_3 (which is the next job in the queue) cannot be started, because there are currently not enough free processors. To enforce the backfilling rule (small jobs can backfill only if, by so doing, they don't delay the first queued job), the scheduler needs to know the runtime of J_2 so as to be able to compute the earliest start time of J_3 (which is T_4). Likewise, the scheduler needs to know that J_4 is short enough so as not delay J_3 if it is backfilled.

To make such determinations possible, users are mandated to provide runtime estimates for each job they submit. And jobs that attempt to exceed their estimates are killed by the system so as not to violate subsequent commitments.

3 Studying the Impact of User Inaccuracy: Wrong Way

The impact of user runtime estimates on the performance of backfilling systems has intrigued many researchers. The first published work we are aware of that investigated the issue was a 1995 technical report from Carnegie Mellon University by Suzuoka et al. [19]; this work came out in the same year as the paper that introduced backfilling [13]. As of this writing, the most recent published work on the subject is an IPDPS 2010 paper by Tang et al. [20], which was awarded best paper (attesting the continued interest in this topic).

These two studies frame 15 years of research (surveyed below) that attempted to understand how inaccurate user estimates affect performance. We argue that the conclusions of most of these research efforts regarding inaccuracy are wrong.

The canonical (and possibly the only) way to study the impact of (in)accuracy of estimates on performance is to: (1) take a workload as depicted in Table 1; (2) manipulate the values within its estimates' column; (3) feed the modified log into a simulator that simulates the run with those artificial estimates; and (4) observe the change in the resulting performance metrics that the simulator outputs. By repeatedly invoking this procedure (initially using completely accurate estimates and then systematically making them less accurate) it is possible to tabulate the performance as a function of the "magnitude" of inaccuracy.

The question is, of course, how to artificially generate those increasingly inaccurate estimates, and how to define and quantify the said magnitude of inaccuracy. We contend that previous studies got this point wrong and that this is why their results are invalid.

The reminder of this Section is dedicated to describing how inaccuracy is typically modeled (Section 3.1); to highlighting the strange, contradictory results such models have yielded and the conflicting attempts to explain them (Section3.2); and to resolving the aforementioned contradiction, while providing the explanation to the counterintuitive results (Section 3.3).

3.1 Modeling Increased Inaccuracy with the *F*-Model

In 1998, to study the sensitivity of backfilling to poor estimates, Feitelson and Mu'alem proposed the "F-model" [8] as follows:

- let r be the runtime of job J,
- let e denote the (artificially-generated) estimate of J,
- let $F \geq 1$ be a "badness factor",
- then e is chosen at random from a uniform distribution $e \in [r, F \cdot r]$.

F was termed the "badness factor", because the artificial estimates start off completely accurate when $F = 0$, and then they become increasingly inaccurate as F grows. Note that, according to the backfilling rules (Section 2.2), F cannot be smaller than 1, since a job must be killed if it tries to exceed its estimate (so as not to violate subsequent commitments), which means $r \leq e$ must always hold.

We note that it is often more convenient to normalize the badness factor so that it would start from zero. We thus set f to be $f = F - 1$ and use the upper- or lowercase notation as is convenient; for the lowercase notation, the following holds:

- $f \geq 0$ (when $f = 0$, the estimates are completely accurate), and
- e is chosen at random from a uniform distribution $e \in [r, (f + 1) \cdot r]$.

The F-model has been used when simulating workloads that lacked estimates data [10,15,25], but, much more importantly, it and its variants have been extensively used to study the impact of inaccurate estimates on backfilling algorithms [1,3,5,8,11,14,17,18,19,20,26,27]; one simpler variant that has been likewise used is the "deterministic F-model" [1,5,15,19,27], in which there is no randomness and each estimate is set to be a direct multiple of the runtime and the badness factor: $e = r \cdot F$.

3.2 The Inaccuracy Mystery

Many of those that utilized the F-model to study inaccuracy reported a surprising, counterintuitive result. They found that inaccurate estimates are usually preferable over accurate ones. This is illustrated in Figure 2 that shows the overall average wait time and bounded slowdown of jobs obtained when simulating the run of the workloads from Table 2 with completely accurate estimates ($F = 1$) and with estimates that are set to be exactly double the runtime ($F = 2$ in the deterministic model). The studies that observed this surprising phenomenon explained it with what we call the *"holes argument"* [1,8,14,15,18,27], as articulated by Chiang et al.:

> ***The Holes Argument:*** *"We note that for large F (or when multiplying estimates by two), jobs with long runtimes can have very large runtime overestimation, which leaves larger 'holes' for backfilling shorter jobs. As a result, average slowdown and wait may be lower."* [1]

Other researchers that utilized the F-model observed a different, though equally counterintuitive, phenomenon. They found that the performance is insensitive to the (in)accuracy of estimates. This is illustrated in Figure 3.[1] Faced with (a tiny fraction of) such results, researchers concluded that the performance is uncorrelated to F [5,11,20,25,27]. For example, England et al. suggested a

[1] In all plots depicting the behavior of the deterministic and the random model along the same X axis, we divide F by 2 in the deterministic case, so as to make both models have the same mean.

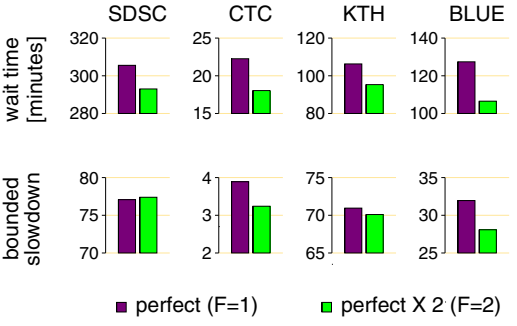


Fig. 2. Lower values mean better performance. Thus, counterintuitively, using completely accurate estimates ("perfect") typically produces inferior results to when estimates are set to be double the runtime ("perfect X 2"). A job's wait time is the duration that elapses between its submission time and the time is starts to run. A job's bounded slowdown is defined to be $\max\left(1\ ,\ \frac{w+r}{\max(10,r)}\right)$, where w and r are the job's wait- and run-times in seconds, respectively; this is a bounded from of the slowdown metric ($\frac{w+r}{r}$) that eliminates the emphasis on very short jobs (shorter than 10 seconds). Performance results throughout this paper are averages across all job.

"robustness" metric for the evaluation of computer systems, and claimed (in a case-study attempting to demonstrate the usefulness of their metric) that:

> ***The Robustness Claim:*** *"Our results support those of a previous work and also indicate that backfilling is robust to inaccurate runtime estimates in general. It seems that, with respect to backfilling, what the scheduler doesn't know won't hurt it."* [5]

Likewise, Tang et al. argued (in their IPDPS'10 best paper) that:

> ***The Insensitivity Claim:*** *"Our analysis indicates that FCFS [with backfilling] is not sensitive to user runtime estimates."* [20]

Those that attempted to explain this surprising finding have done so with the help of what we call the "balance argument" [11,25,26,27], as articulated by Zhang et al.:

> ***The Balance Argument:*** *"We can understand why backfilling is not that sensitive to the estimated execution time by the following reasoning. On average, overestimation impacts both the jobs that are running and the jobs that are waiting. The scheduler computes a later finish time for the running jobs, creating larger holes in the schedule. The larger holes can then be used to accommodate waiting jobs that have overestimated execution times. The probability of finding a backfilling candidate effectively does not change with the overestimation."* [25]

For example, doubling the lengths of all the jobs in Figure 1 only means the X-axis is scaled by a factor of two, but doesn't change anything regarding the backfilling

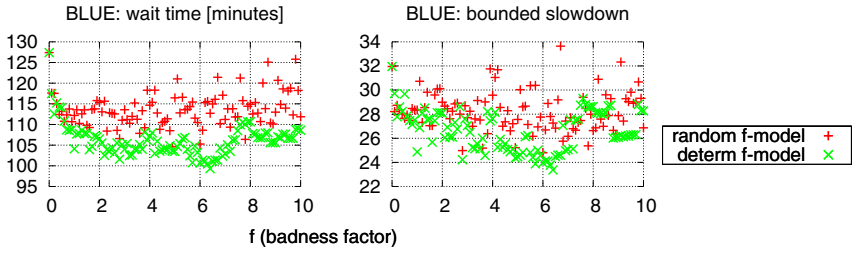


Fig. 3. Performance as a function of the badness factor f, using the random and the deterministic f-models, with a resolution of 0.1 (that is, $f = 0, \frac{1}{10}, \frac{2}{10}, \frac{3}{10}, ..., 10$). Counterintuitively, there is no clear connection between f and the associated performance.

decision: indeed, after doubling, job 4 looks twice as long in the eyes of the scheduler, but the same applies to the 2-time-units-hole opened by job 2, so job 4 can backfill before the doubling if and only if it can do so after the doubling.

While both the holes argument and the balance argument seemingly make sense, one obvious problem with them is that they are contradictory. If the balance argument is correct, then there is no benefit in opening those "larger holes" as suggested by the holes argument, because backfilling candidates would become proportionally longer and cancel the effect. Conversely, the holes argument implies a performance improvement that is proportional to F, in contrast to the balance argument rationale. Indeed, the holes argument seems contradictory to Figure 3, and the balance argument seems contradictory to Figure 2.

3.3 Solving the Mystery: The Heel-and-Toe Backfilling Dynamic

To make sense of the counterintuitive, contradictory findings, we do what inaccuracy studies should have done in the first place but for some reason didn't. Namely, we exploit the random component of the F-model in order to quantify performance in terms of statistical mean and confidence intervals. As it turns out, doing so transforms the noisy results (presented in Figure 3) into well-behaved curves that expose a clear trend as demonstrated in Figure 4.

The fact that a clear trend exists means that all the papers that argued that performance is insensitive to accuracy were mistaken. Their mistake was caused by conducting only a few experiments (a tiny fraction of Figure 3), instead of achieving statistical confidence.

Three of the four simulated logs (SDSC and KTH not shown) produce results similar to that of BLUE as depicted in the left of Figure 4. The performance trend for these logs can be characterized as "V shaped", namely, initially the curves drop (performance improves) and then the trend is reversed (performance worsens). The performance of CTC is "L shaped", asymptotically converging to some value after the initial drop.

We will now explain why the curves behave as they do, starting with the initial drop that indicates performance improvement across all four logs for small

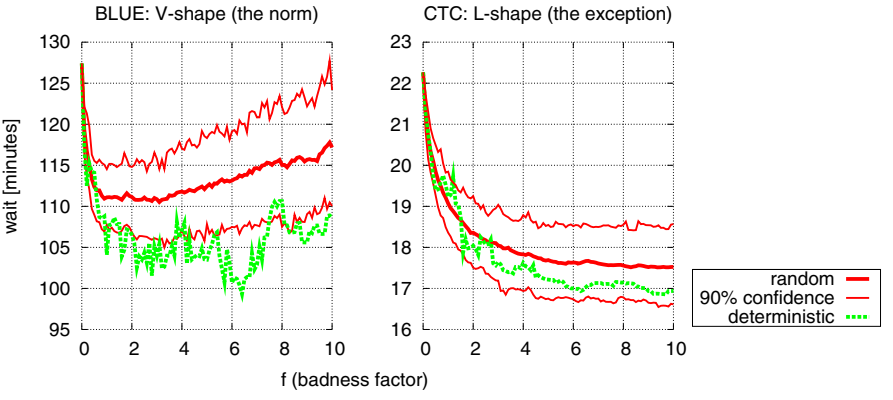


Fig. 4. Averaging multiple experiments exposes a clear trend (compare with Figure 3). For every f (where $f = \frac{1}{10}, \frac{2}{10}, \frac{3}{10}, ..., 10$), we conduct 100 simulations with different seeds; the "random" curve shows the mean of these runs, and the matching "90% confidence" curves show the 5th percentile to the 95th percentile. (We thus performed $100^2 = 10,000$ simulations of scheduling the jobs for each trace.) The deterministic model is unaffected by different seeds (lacking a random component), and so the associated results remain noisy, but we can see that the deterministic curve roughly approximates the best case scenario of the random experiments. Although not shown, the SDSC and KTH logs produce qualitatively similar results to that of BLUE (for both bounded slowdown and wait time) [24].

f values. We begin by noting that, in accordance to the holes argument (and in contrast to the balance argument), backfilling activity intensifies when inaccuracy is increased as shown in Figure 5. Namely, the larger f is, the more jobs enjoy backfilling.

The question is why? What's wrong with the balance argument? Why aren't the bigger holes canceled out by the proportionally bigger backfill candidates? The answer is the "heel-and-toe" backfilling dynamic,[2] which we illustrate in Figure 6 and characterize next. For simplicity, we assume all estimates are exactly double the runtime (F=2 under the deterministic model). Based on the information available to the scheduler at T_0 (time 0), it appears the earliest time for J_3 (job 3) to start is T_{12}, even though the real earliest start time is actually T_6. Thus, the scheduler makes a "reservation" on J_3's behalf for T_{12} and can only backfill jobs that honor this reservation. At T_4, J_2 terminates. As J_1 is still running, nothing has changed with respect to J_3's reservation, and so the scheduler scans the wait queue in search of appropriate candidates for backfilling. J_4 (the first backfill candidate under FCFS) fits the gap between T_4 and the reservation (T_{12}) and it is therefore backfilled, effectively pushing back the real earliest time at which J_3 could have started from T_6 to T_8. (Likewise, when J_1 terminates,

[2] In the Talmud, the expression "heel-and-toe" describes a slow and careful motion, whereby a person advances by repeatedly moving the heel of the back foot adjacent to the toe of the front foot.

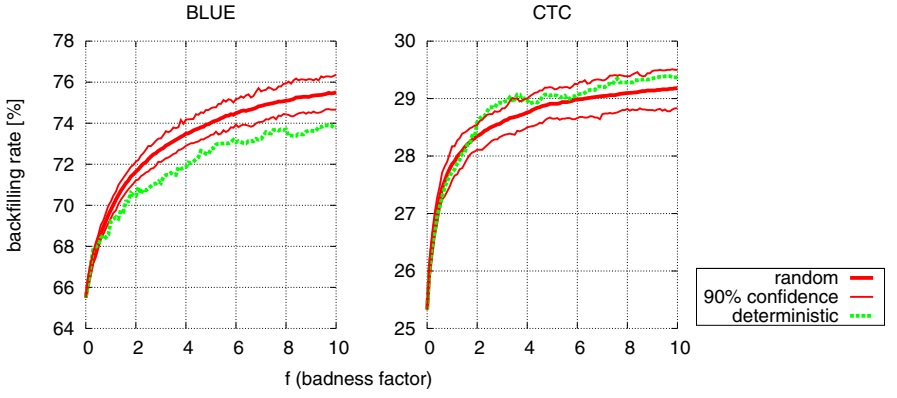


Fig. 5. The percent of backfilled jobs monotonically increases with f

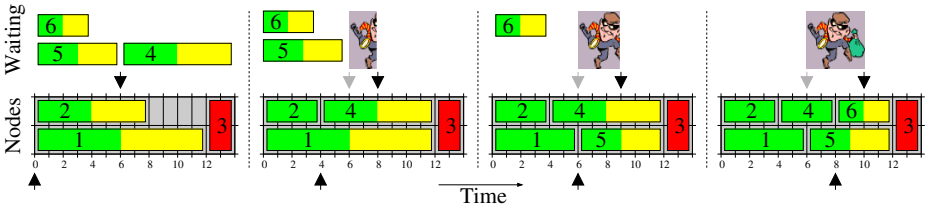


Fig. 6. Illustrating the heel-and-toe backfilling dynamic. Job numbers indicate arrival order. Job estimates are exactly double their runtime (F=2). The left portion of jobs (green/dark) indicates their real runtimes. Due to the doubling, the scheduler views jobs as twice as long (right portion; yellow/bright). The bottom arrows show the progress of time, whereas the top black arrows show the earliest time at which job 3 would have been started, had real runtimes been known to the scheduler (at that point in time). The thief's width shows the amount of "stolen" time, at the expense of job 3.

then J_5 is backfilled, and when J_4 terminates, then J_6 is backfilled, respectively pushing J_3's real earliest start time to T_9 and then T_{10}.)

To exemplify that the heel-and-toe dynamic does indeed occur, we define "wild backfilling" to be a backfill decision that result in a delay of the earliest start time possible of the first queued job (all backfill decision in Figure 6 are wild). We further define the "stolen time" to be the duration of the time interval by which the earliest start time got delayed (in Figure 6 this is 4 time units, from T_6 to T_{10}). Figure 7 confirms that the heel-and-toe dynamic does in fact occur, by showing the wild backfilling rate and average stolen time within the SDSC simulations (again, the other logs are similar).

The heel-and-tow dynamic induces a state whereby shorter jobs (those that fit the steadily shrinking holes) are favored. This explains the performance improvement. In particular, Figure 7 shows that the dynamic intensifies as F grows, explaining the observed performance improvement trend caused by steadily growing inaccuracy (initial part of the V and L curves in Figure 4).

Importantly, notice that the heel-and-tow dynamic reconciles between the contradictory holes argument and balance argument. The performance improvement attributed to positive Fs is not because of wider holes in the schedule that allow for more backfilling (in accordance to the holes argument), because backfill candidates are indeed widened proportionally (in accordance to the balance argument). Rather, it is the result of a heel-and-toe effect that manages to *keep the holes open* by backfilling shorter jobs in a way that repeatedly delays the execution of the first queued job.

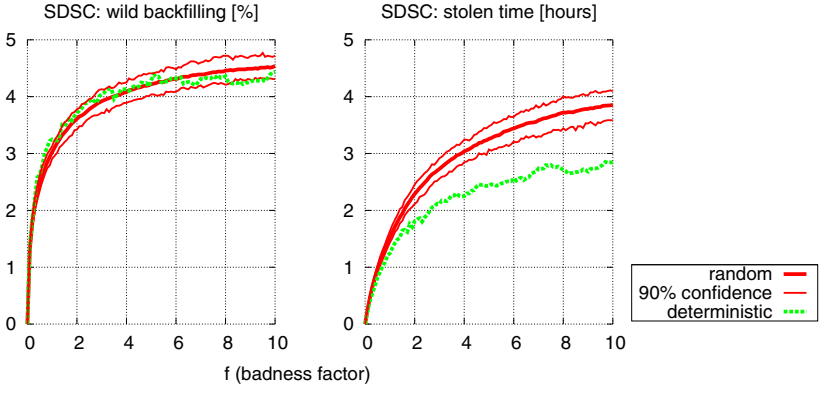


Fig. 7. The impact of the heel-and-toe backfilling dynamic. In SDSC, up to 5% of the jobs are started as a result of a wild backfilling decision (left), causing the first queued jobs to be delayed by up to 4 hours (right), on average.

To finish, we need to explain why performance worsens for all but the CTC log (the ascending, right part of the V shape) and why CTC is different (L-shaped). The explanation is detailed elsewhere [21,24], and, to keep the discussion focused, we only provide the intuition here. The performance is worsened, because the probability that the scheduler will "mistake" short jobs for long (and vice versa) monotonically increases with F. Formally, if the runtime of two jobs J_1 and J_2 is r_1 and r_2, respectively, and we assume (without loss of generality) that $r_1 < r_2$, then we can prove that the probability $Pr(e_1 > e_2)$ monotonically increases with F, where $e_1 \in [r_1, r_1 \cdot F]$ and $e_2 \in [r_2, r_2 \cdot F]$ are the randomly chosen estimates of J_1 and J_2, respectively. (And this is, by the way, the reason why the deterministic curve roughly follows the best case scenario of the random model in Figure 4, as this probabilistic argument doesn't apply.) The reason CTC is largely unaffected by the probabilistic argument, is that it lacks the bursty nature that the other workloads have (meaning, the wait queue is typically short and so the chances of making the $Pr(e_1 > e_2)$ mistake are smaller); when artificially introducing burstiness to CTC, the associated performance curves become V-shaped like that of the other workloads.

4 Studying the Impact of User Inaccuracy: Right Way

The previous section might seem to suggest that increased inaccuracy improves performance (thereby corroborating the conclusions of many past studies). Nothing can be further from the truth. The conclusions of the previous section are based on *artificial* inaccuracy as generated by the F-model, whereas *real* inaccuracy (as manifested by real users) is inherently different, and it affects performance in an entirely different way. In reality, less accurate estimates worsen the performance, in contrast to what we've learned in the previous section [23].

As it turns out, inaccuracy of human users takes the form of utilizing very few "round" values, such as 15 minutes, 1 hour, and oftentimes the maximal runtime allowed. In fact, in all of our logs, merely 20 such canonical values are used by 90% of the jobs as estimates. This user behavior is clearly evident from Figure 8, which plots the cumulative distribution function of the runtimes and estimates of jobs. In contrast to the smooth runtime curve, the estimates form a rigid staircase-like structure, whereby each stair is associated with a single popular estimate value.

Importantly, the modality of estimates hampers backfilling systems, because jobs with different runtimes all look the same to the scheduler, preventing it from distinguishing between short and long jobs and limiting its ability to utilize existing holes in the schedule. (Conversely, by definition, the F-model provides a fairly good relative ordering of the jobs and a lot of variability for the scheduler to work with.)

Especially harmful to performance is the fact that the maximal allowed runtime is always a very popular estimate value among users (e.g., in the case of

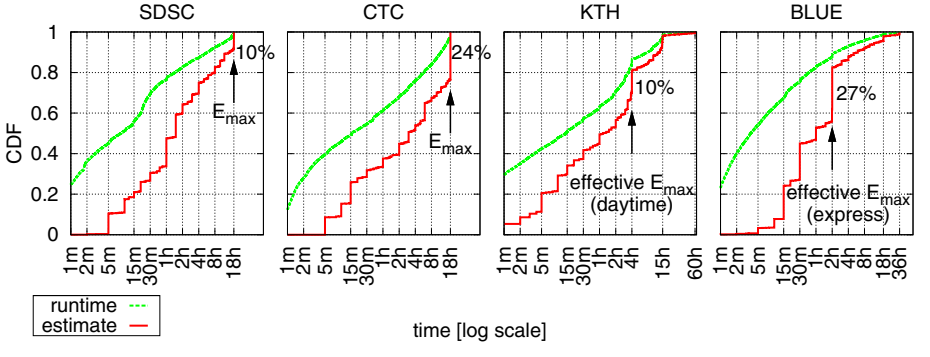


Fig. 8. Cumulative distribution function (CDF) of jobs' runtimes and estimates. The runtime curves appear higher because runtimes are always shorter than estimates (underestimated jobs are killed). We denote the maximal allowed runtime as E_{max}. (This is also the maximal allowed estimate, as jobs are allowed to run until their estimate is reached.) In SDSC and CTC the E_{max} is 18h; in KTH and BLUE, 4h and 2h serve as the "effective" E_{max}, because most jobs were submitted during daytime or to the express queue, respectively, and 4h and 2h are the associated limits enforced on those systems. Clearly, E_{max} is a popular value.

CTC, 25% of the jobs utilized this value as estimate; see Figure 8). It is harmful because such jobs are *never* backfilled. Indeed, if all the jobs choose the maximum as their estimate, then there would be no backfilling activity, and the schedule would largely revert to plain FCFS.

The bottom line is that, when researchers wish to assess the impact of inaccuracy on performance, they should *not* use the F-model, as using it for this purpose constitutes a serious methodological error that would likely invalidate their results [23]. When using the F-model, researchers simply convey to the scheduler too much information that it would probably never enjoy in reality (fairly accurate relative ordering of jobs), and they additionally induce the heel-and-toe dynamic, which further unrealistically improves the results of their evaluation by artificially favoring shorter jobs.

The correct way to evaluate the impact of increased inaccuracy is by making the estimates distribution more modal. (For example, by associating an increasing number of jobs with the maximal runtime allowed.) In a different work, we have developed a detailed model that accurately captures the modal nature of the estimates distribution and that allows its users to control the "amount" of modality [23]. The model is freely available for download [22] from the parallel workload archive [16].

5 Conclusions

Backfilling drastically improves the system utilization [12] by allowing jobs to run ahead of their time, provided they do not delay higher-priority jobs. But in order to do so, backfill systems require users to estimate how long their jobs would run. Ever since the inception of backfilling, researchers wondered about the impact of inaccurate estimates on performance, and many studies addressed this issue (surveyed above).

To evaluate the impact of inaccuracy, researchers associated each job with an artificial estimate e that is a multiple of the actual runtime r with some "badness" factor F, such that $e = r \cdot F$ (or such that e is correlated with $r \cdot F$); with this "F-model", larger Fs supposedly imply increased inaccuracy. Relying on the F-model, researchers repeatedly reached a counterintuitive conclusion: that performance improves by (or is insensitive to) increased inaccuracy.

In this paper we have refuted this counterintuitive conclusion, by exposing the F-model to be erroneous. It artificially conveys too much information to the scheduler (the relative ordering of jobs), and, in addition, it implicitly nudges the system towards shortest-job scheduling through a "heel-and-toe" dynamic that manages to keep backfilling windows open at the expense of the first-queued job. In contrast, the inaccuracy of real user estimates worsens the performance, because users utilize very few "round" estimates (especially the maximal runtime), making it hard for the scheduler to distinguish between long and short jobs and hindering its ability to backfill effectively (e.g., jobs with the maximal runtime as estimate would never be backfilled).

We thus proclaim that the popular F-model is inappropriate for being used in studies that wish to learn the effect of inaccurate user estimates. Researchers

should stop using multiples of actual runtimes as estimates, or else they would likely get invalid results. To get trustworthy results, researchers should preserve the modal nature of user estimates [23]. We have made available a model that does so [22], and we recommend to prefer it over the F-model; with the suggested model, researchers can explore the impact of increased user inaccuracy by, e.g., increasingly associating more estimates with the maximal runtime allowed.

References

1. Chiang, S.-H., Arpaci-Dusseau, A., Vernon, M.K.: The impact of more accurate requested runtimes on production job scheduling performance. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2002. LNCS, vol. 2537, pp. 103–127. Springer, Heidelberg (2002)
2. Chiang, S.-H., Vernon, M.K.: Production job scheduling for parallel shared memory systems. In: 15th IEEE Int'l Parallel & Distributed Processing Symp (IPDPS) (April 2001)
3. Dimitriadou, S., Karatza, H.: Job scheduling in a distributed system using backfilling with inaccurate runtime computations. In: IEEE Int'l Conf. Complex, Intelligent & Software Intensive Systems (CISIS), pp. 329–336 (February 2010)
4. Dongarra, J.J., Meuer, H.W., Simon, H.D., Strohmaier, E.: Top500 supercomputer sites, http://www.top500.org/ (updated every 6 months)
5. England, D., Weissman, J., Sadago-pan, J.: A new metric for robustness with application to job scheduling. In: 14th IEEE Int'l Symp. on High Performance Distributed Comput. (HPDC), pp. 135–143 (July 2005)
6. Ernemann, C., Krogmann, M., Lepping, J., Yahyapour, R.: Scheduling on the top 50 machines. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2004. LNCS, vol. 3277, pp. 17–46. Springer, Heidelberg (2005)
7. Etsion, Y., Tsafrir, D.: A Short Survey of Commercial Cluster Batch Schedulers. Technical Report 2005-13, The Hebrew University of Jerusalem (May 2005)
8. Feitelson, D.G., Mu'alem Weil, A.: Utilization and predictability in scheduling the IBM SP2 with backfilling. In: 12th IEEE Int'l Parallel Processing Symp (IPPS), pp. 542–546 (April 1998)
9. Feitelson, D.G., Rudolph, L., Schwiegelshohn, U.: Parallel job scheduling — a status report. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2004. LNCS, vol. 3277, pp. 1–16. Springer, Heidelberg (2005)
10. Frachtenberg, E., Feitelson, D.G., Petrini, F., Fernandez, J.: Adaptive parallel job scheduling with flexible coscheduling. IEEE Trans. on Parallel & Distributed Syst. (TPDS) 16(11), 1066–1077 (2005)
11. Guim, F., Corbalán, J., Labarta, J.: Prediction f based models for evaluating backfilling scheduling policies. In: 8th IEEE Int'l Conf. on Parallel & Distributed Computing, Applications & Technologies (PDCAT), pp. 9–17 (December 2007)
12. Jones, J.P., Nitzberg, B.: Scheduling for parallel supercomputing: a historical perspective of achievable utilization. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 1999, IPPS-WS 1999, and SPDP-WS 1999. LNCS, vol. 1659, pp. 1–16. Springer, Heidelberg (1999)
13. Lifka, D.: The ANL/IBM SP scheduling system. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1995 and JSSPP 1995. LNCS, vol. 949, pp. 295–303. Springer, Heidelberg (1995)

14. Mu'alem, A., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. IEEE Trans. on Parallel & Distributed Syst (TPDS) 12(6), 529–543 (2001)
15. Netto, M.A.S., Buyya, R.: Coordinated Rescheduling of Bag-of-Tasks for Executions on Multiple Resource Providers. Technical Report CLOUDS-TR-2010-1, U. of Melbourne, Australia, Submitted (TPDS) (February 2010)
16. Parallel Workloads Archive, http://www.cs.huji.ac.il/labs/parallel/workload
17. Sabin, G., Sadayappan, P.: On enhancing the reliability of job schedulers. In: High Availability & Performace Computing Workshop (HAPCW) (October 2005)
18. Srinivasan, S., Kettimuthu, R., Subrarnani, V., Sadayappan, P.: Characterization of backfilling strategies for parallel job scheduling. In: Int'l Conf. on Parallel Processing (ICPP), pp. 514–522 (August 2002)
19. Suzuoka, T., Subhlok, J., Gross, T.: Evaluating Job Scheduling Techniques for Highly Parallel Computers. Technical Report CMU-CS-95-149, School of Computer Science, Carnegie Mellon University (August 1995)
20. Tang, W., Desai, N., Buettner, D., Lan, Z.: Analyzing and adjusting user runtime estimates to improve job scheduling on the Blue Gene/P. In: IEEE Int'l Parallel & Distributed Processing Symp (IPDPS) (April 2010)
21. Tsafrir, D.: Modeling, Evaluating, and Improving the Performance of Supercomputer Scheduling. PhD thesis, The Hebrew University of Jerusalem (September 2006)
22. Tsafrir, D., Etsion, Y., Feitelson, D.G.: A model/utility for generating user runtime estimates and appending them to a standard workload format (SWF) file (February 2006), http://www.cs.huji.ac.il/labs/parallel/workload/m_tsafrir05
23. Tsafrir, D., Etsion, Y., Feitelson, D.G.: Modeling user runtime estimates. In: Feitelson, D.G., Frachtenberg, E., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2005. LNCS, vol. 3834, pp. 1–35. Springer, Heidelberg (2005)
24. Tsafrir, D., Feitelson, D.G.: The dynamics of backfilling: solving the mystery of why increased inaccuracy may help. In: 2nd IEEE Int'l Symp. on Workload Characterization (IISWC) (October 2006)
25. Zhang, Y., Franke, H., Moreira, J., Sivasubramaniam, A.: Improving parallel job scheduling by combining gang scheduling and backfilling techniques. In: 14th IEEE Int'l Parallel & Distributed Processing Symp. (IPDPS), pp. 133–142 (May 2000)
26. Zhang, Y., Franke, H., Moreira, J., Sivasubramaniam, A.: An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration. IEEE Trans. on Parallel & Distributed Syst. (TPDS) 14(3), 236–247 (2003)
27. Zotkin, D., Keleher, P.J.: Job-length estimation and performance in backfilling schedulers. In: 8th IEEE Int'l Symp. on High Performance Distributed Comput. (HPDC), p. 39 (August 1999)

Author Index