Cost-Minimizing Scheduling of Workflows on a Cloud of Memory Managed Multicore Machines

Nicolas G. Grounds[1], John K. Antonio[2], and Jeff Muehring[1]

[1] RiskMetrics Group, 201 David L. Boren Blvd, Suite 300, Norman, OK, USA
[2] School of Computer Science, University of Oklahoma, Norman, OK, USA

Abstract. Workflows are modeled as hierarchically structured directed acyclic graphs in which vertices represent computational tasks, referred to as requests, and edges represent precedent constraints among requests. Associated with each workflow is a deadline that defines the time by which all computations of a workflow should be complete. Workflows are submitted by numerous clients to a scheduler that assigns workflow requests to a cloud of memory managed multicore machines for execution. A cost function is assumed to be associated with each workflow, which maps values of relative workflow tardiness to corresponding cost function values. A novel cost-minimizing scheduling framework is introduced to schedule requests of workflows so as to minimize the sum of cost function values for all workflows. The utility of the proposed scheduler is compared to another previously known scheduling policy.

1 Introduction

The service-oriented architecture (SOA) framework is a viable approach to cloud computing in which computational requirements of a user are represented by basic service requests. In this framework, the computational requirements of a user are modeled as a workflow graph (WFG), which is a directed and acyclic graph that defines precedence constraints among service requests required by the user. WFGs can vary greatly in size and structure. For example, a small WFG may contain just a few requests (i.e., vertices) while a large WFG may contain thousands of requests. Regarding structure, at one extreme a WFG may represent a single chain of requests in which no two requests may be executed in parallel. At another extreme, the structure of a WFG may contain numerous independent chains of requests in which requests belonging to distinct chains may be executed in parallel.

For the purposes of this paper, the SOA is supported by a collection of memory-managed multicore machines. Each machine supports one or more services, and associated with each service are a number of supporting operations. A service request involves the execution of an operation provided by a service. Each multicore machine in the assumed platform can concurrently execute multiple service requests because each request is executed as an independent thread on the machine. The instantaneous performance efficiency of each machine is assumed to depend on an aggregate measure of CPU loading and heap memory loading

M.G. Jaatun, G. Zhao, and C. Rong (Eds.): CloudCom 2009, LNCS 5931, pp. 435–450, 2009.

of all requests executing on the machine. An efficiency-based performance model for memory-managed multicore machines is adopted in this paper.

In the framework considered here, WFGs are assumed to be submitted by multiple clients to a scheduler. Associated with each submitted WFG is a deadline that defines the time by which all requests of the WFG should complete execution. A cost function is assumed to be associated with each workflow, which maps values of workflow tardiness to corresponding cost function values. A novel cost-minimizing scheduling approach is introduced to schedule requests of workflows so as to minimize the sum of cost function values for all workflows.

The remainder of the paper is organized in the following manner. Section 2 includes an overview of related work. Section 3 describes the assumed cloud environment, including models and descriptions for the workflow graphs and the machines that support the cloud's SOA. Section 4 describes the new cost-minimizing scheduler. Section 5 provides the results of simulation studies, followed by concluding remarks in the final section.

2 Background and Related Work

Previous related work is reviewed in three broad areas: (1) machine modeling and simulation environments; (2) automatic memory management; and (3) scheduling and load balancing.

Considerable work has been published related to modeling of machines in distributed environments. Much of the past research in this area has focused on modeling and predicting CPU performance, e.g., [1, 2]. The machine model described in the present paper (refer to Section 3.3) relies on assumed knowledge of the characteristics of the requests (i.e., computational tasks); it is similar in a sense to the SPAP approach proposed in [1].

In memory managed systems, the effect of long and/or frequent garbage collections can lead to undesirable – and difficult to predict – degradations in system performance. Garbage collection tuning, and predicting the impact of garbage collections on system performance, are important and growing areas of research, e.g., [3, 4, 5, 6, 7]. To estimate the overhead associated with various garbage collectors, experiments were designed and conducted in [4, 5] to compare the performance associated with executing an application assuming automatic memory management versus explicit memory management. The machine model proposed here accounts for the overhead associated with automatic memory management.

Formulations of realistic scheduling problems are typically found to be NP-complete, hence heuristic scheduling policies are generally employed to provide acceptable scheduling solutions, e.g., refer to [7, 8, 9, 10, 11, 12]. The scheduling evaluations conducted in the present paper account for the impact that garbage collection has on a machine's performance. Examples of other memory-aware scheduling approaches are described in [7, 13].

Load balancing involves techniques for allocating workload to available machine(s) in a distributed system as a means of improving overall system performance. Examples of both centralized and distributed load balancing approaches

are described in [8]. The scheduling framework developed in the present paper incorporates load balancing in the sense that the scheduler only assigns requests to machines if doing so is beneficial relative to minimizing a desired cost function. For simplicity the algorithm assumes every machine can service any request although relaxing this assumption is a straight-forward extension to the approach described here.

3 Cloud Environment

3.1 Overview

Fig. 1 illustrates the major components of the assumed system in which clients submit workflow graphs (WFGs) to a cloud environment for execution. Clients purchase computing services from the cloud, which implements a service-oriented architecture. Associated with each WFG is a service-level agreement (SLA) [9] that defines a deadline for finishing all computations of the WFG. An SLA generally defines cost penalties in the event that the terms of an SLA are not met, e.g., a deadline is missed. For instance, a cost penalty value increases as a function of increasing WFG tardiness. The precise terms of an SLA are carefully constructed for business applications in which timely delivery of computational results are a critical component of a client process — and not having these results delivered to the client by the deadline incurs costs.

The next two subsections provide descriptions and models for the WFG and machine components shown in Fig. 1. Variants of the material presented in Subsections 3.2 and 3.3 were originally introduced in [14]. A primary contribution of the present paper is the introduction of the new cost-minimizing scheduler, which is described in Section 4 and evaluated through simulation studies in Section 5.

3.2 Workflow Graph (WFG) Model

A WFG is a directed acyclic graph with a hierarchical structure composed of parallel and sequential combinations of request chains (RCs). An example WFG is shown in Fig. 2(a). The vertices of the graph represent requests and the directed arcs denote precedence constraints that exist between requests, e.g., request 2 in Fig. 2(a) cannot begin executing until request 1 finishes executing.

The hierarchical nature of the WFG of Fig. 2(a) is illustrated by the tree structure in Fig. 2(b). The leaf nodes of the tree represent the requests of the WFG. Traversing the tree in a depth-first order defines the structure of the associated WFG; non-leaf tree nodes are labeled "S" or "P," which defines whether that node's children must be executed sequentially (S) or may be executed in parallel (P). The children nodes (sub-trees) of a node labeled P are assumed to represent independent and identical computational structures executed with distinct input data. Although all of the children sub-trees of a P node could potentially be executed in parallel, it may not be possible (or effective) to fully

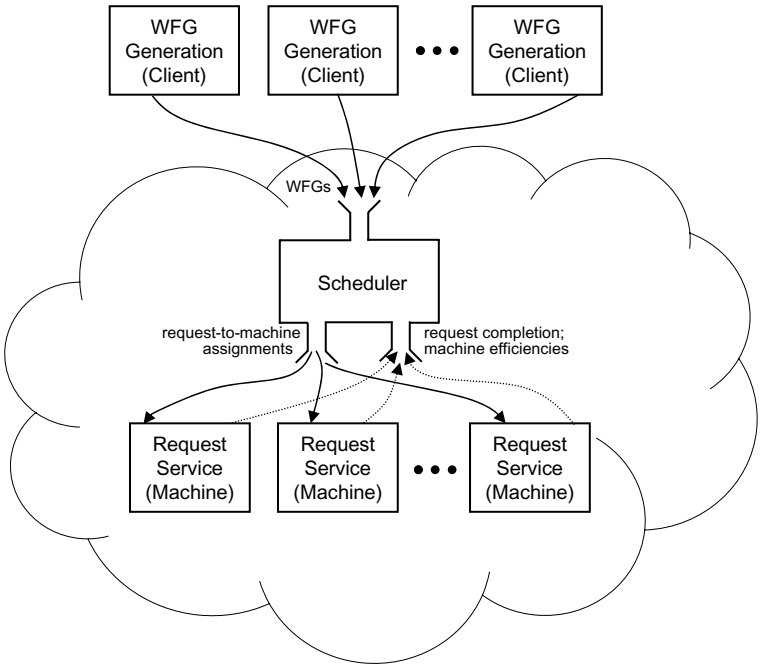


Fig. 1. Major components of the system model

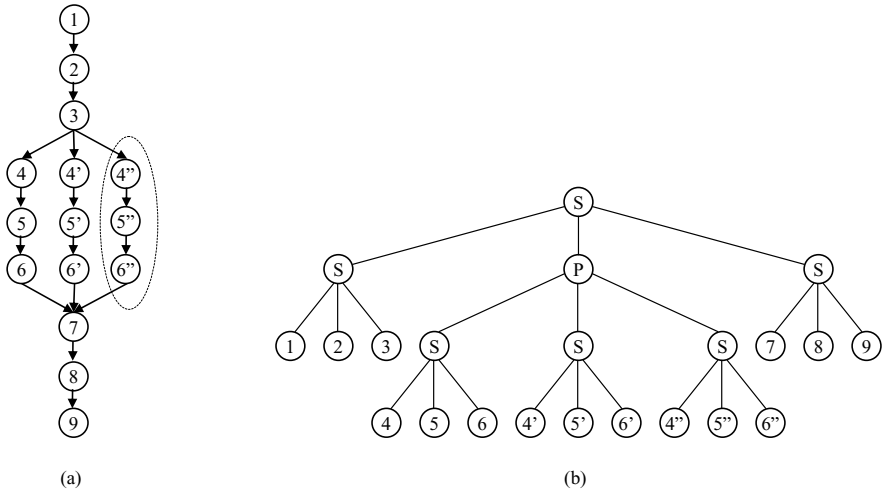


Fig. 2. (a) Sample WFG with one of five RCs encircled. (b) Hierarchical structure of the WFG shown in (a).

exploit all available parallelism associated with all currently executing WFGs due to resource limitations and/or loading.

When a WFG arrives at the scheduler, it is placed in a pool that holds all WFGs that have not yet finished execution. Once all requests of a WFG have finished execution, the entire WFG is defined as finished and removed from the scheduling pool. The scheduler tracks the status of individual requests according to the following states: "blocked," "ready," "executing," or "finished."

The time instant that the state of a request r transitions from "blocked" to "ready" is defined as r's birth time and is denoted by b_r. The finish time of a request is denoted by f_r. The birth time of a request r is defined as the maximum of all finish times of r's precedence requests. For example, in Fig. 2(a), $b_7 = \max\{f_6, f_{6'}, f_{6''}\}$. The time instant that the state of a request transitions from "ready" to "executing" is defined as the request's start time and is denoted by s_r. The start time of a request must be greater than or equal to its birth time, i.e., $s_r \geq b_r$. The function of the scheduler is to determine the start time s_r for each request r as well as determine r's machine assignment, denoted by M_r.

The time instant when WFG $\boldsymbol{w}$ arrives at the scheduling pool is defined as $\boldsymbol{w}$'s birth time, denoted by $b_{\boldsymbol{w}}$. The birth time of a WFG is also the birth time of all requests in the WFG that have no precedence constraints, e.g., the birth time of the WFG in Fig. 2(a) equals the birth time of request 1. The start time of a WFG is defined as the minimum start time value of all requests associated with the WFG. Thus, the start time of $\boldsymbol{w}$ is defined by $s_{\boldsymbol{w}} = \min_{r \in \boldsymbol{w}}\{s_r\}$. The finish time of $\boldsymbol{w}$, denoted $f_{\boldsymbol{w}}$, is defined as the maximum finish time of all requests in $\boldsymbol{w}$, i.e., $f_{\boldsymbol{w}} = \max_{r \in \boldsymbol{w}}\{f_r\}$.

Associated with each WFG $\boldsymbol{w}$ is a known deadline $d_{\boldsymbol{w}}$, which defines the point in time by which $\boldsymbol{w}$ should finish execution. If $f_{\boldsymbol{w}} \leq d_{\boldsymbol{w}}$, then $\boldsymbol{w}$ is not tardy; otherwise (if $f_{\boldsymbol{w}} > d_{\boldsymbol{w}}$) $\boldsymbol{w}$ is declared to be tardy. By making judicious choices for request start times and machine assignments, the scheduler attempts to minimize the cost associated with workflow tardiness. Because each machine has a finite capacity, assigning too many concurrent requests to the same machine can degrade the efficiency of that machine, thus extending the finish times of all requests assigned to that machine. Extending the finish times of requests can ultimately extend the finish time of the corresponding WFGs, possibly leading to one or more being tardy.

3.3 Efficiency-Based Machine Model

Each request is assumed to require a fraction of two basic resources available on each machine of the cloud: CPU cycles and heap memory. Table 1 summarizes the notation and definitions of basic computational and heap memory requirements for request r.

The CPU utilization factor of r, U_r, can be no greater than unity and no less than zero. A request having a CPU utilization factor of unity is typically referred to as a CPU-bound request, e.g., refer to [1].

The efficiency value for a machine depends on the aggregate CPU and heap memory loading due to all requests executing on the machine. The CPU and heap

Table 1. Definitions of CPU and heap memory requirements for request r

$C_r > 0$	C_r is the number of CPU cycles required to complete r.
$I_r \geq C_r$	I_r is the execution time duration of r on an ideal machine.
$U_r = C_r/I_r$	U_r is the CPU utilization factor of r.
$H_r > 0$	H_r is the maximum reachable heap memory requirement of r.

memory loading of a given machine changes with time only when new requests are assigned and start executing on the machine, or when existing requests finish execution on the machine. Generally, The efficiency value of a machine generally decreases when new requests begin executing on the machine, and increases when request(s) complete execution on that machine.

The machine to which request r is assigned is denoted by M_r. The efficiency of machine M_r from time instance t_i to time instance t_{i+1}, denoted by $e(M_r, t_i)$, has a value between zero and unity. The number of CPU cycles remaining to complete execution of request r at time instance t_i is denoted by $c_r(t_i)$. The value of $c_r(t_{i+1})$ is calculated based on $c_r(t_i)$ according to the following equation:

$$c_r(t_{i+1}) = \begin{cases} C_r, & t_{i+1} < s_r \\ \max\{0, c_r(t_i) - (t_{i+1} - t_i)e(M_r, t_i)U_r\}, & t_{i+1} \geq s_r \end{cases} \quad (1)$$

For time instants less than r's start time, the value of $c_r(t)$ remains constant at C_r (see Table 1) because the request has not yet started executing. For time instants greater than the request's start time, the value of $c_r(t)$ decreases according to the difference equation defined by the second case of Eq. 1. The value deducted from the CPU cycles remaining to complete execution of request r is proportional to the product of the efficiency of the machine on which the request is assigned and that request's CPU utilization factor. Thus, the maximum possible deduction is $t_{i+1} - t_i$, which corresponds to a situation in which the request is executing on a machine with an efficiency of unity and the request has a CPU utilization factor of unity. The application of the max function in the equation ensures that the number of CPU cycles remaining to complete execution of request r is non-negative.

Fig. 3 illustrates how changes in a machine's efficiency value affects the time required to execute a request on that machine. From the figure, notice that request r starts executing on the assigned machine at $t = s_r$. Near the beginning of the request's execution, note that the efficiency of the machine is relatively high, and the slope of the curve for $c_r(t)$ is correspondingly steep (refer to Eq. 1). Throughout the execution of request r, other requests start executing on the machine (corresponding to decreases in the machine's efficiency value) and complete execution on the machine (corresponding to increases in the machine's efficiency value). The finish time of r is defined as the first point in time when $c_r(t) = 0$, indicated by f_r in Fig. 3.

The following discussion describes how the value of a machine's efficiency is modeled. Throughout this discussion, it is understood that the efficiency value is related to a particular machine for a particular time instant. Thus, the value

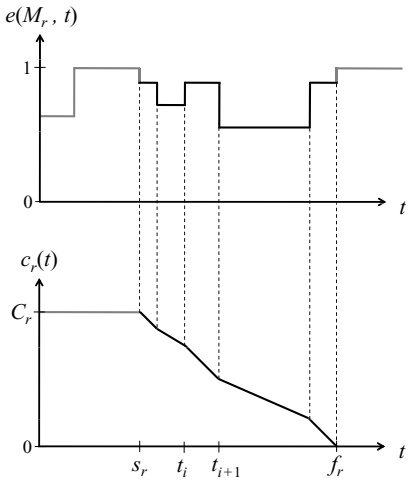


Fig. 3. Illustration of how a machine's efficiency value affects the time required to execute a request on the machine

of efficiency is often referred to as simply e, instead of $e(M, t)$, to ease notational burden.

CPU loading and heap memory loading are the two primary factors used to characterize a machine's relative efficiency. In the machine model, the overall efficiency of a machine is defined by the product of two terms:

$$e = e_c e_h. \tag{2}$$

The terms on the right hand side of Eq. 2 are defined as the CPU efficiency and heap efficiency, respectively. The values of e_c and e_h represent the relative impact on a machine's overall efficiency due to loading of the machine's CPU and heap resources, respectively. The specific functions assumed in the present paper are given by Eq. 3 and Eq. 4.

$$e_c = \begin{cases} 1, & \ell_c < 4 \\ (4/\ell_c), & \ell_c \geq 4 \end{cases} \tag{3}$$

$$e_h = \frac{10}{10 + \frac{1}{(1/\ell_h)-1}} \tag{4}$$

Derivations of these two functions are provided in [14]. The CPU efficiency function of Eq. 3 models a quad-core machine with a CPU loading factor of $\ell_c \geq 0$. The value of ℓ_c is assumed to equal the sum of the U_r's (CPU utilization factors) of all requests executing on the machine. The heap efficiency function of Eq. 4 models the efficiency of the machine's memory managed system as a function of a normalized heap loading factor, $0 < \ell_h < 1$. The specific function of Eq. 4 assumes the time required for a single (full) garbage collection is 10 times less than the execution time of the typical request execution time.

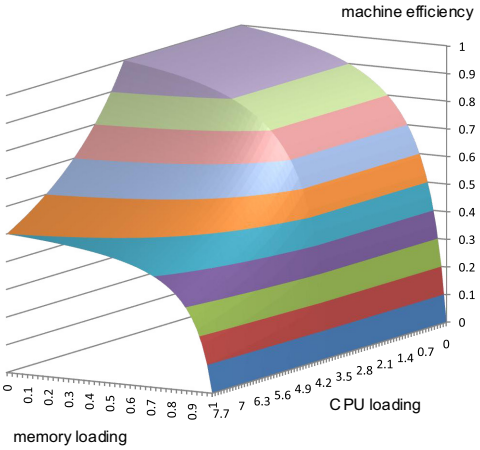


Fig. 4. Derived machine efficiency surface based on the functions for e_c in Eq. 3 and e_h in Eq. 4

Fig. 4 shows a two-dimensional surface plot of $e = e_c e_h$, which is the product of the formulas given in Eqs. 3 and 4. This efficiency function surface is assumed for each machine for the simulations conducted in Section 5.

4 Cost-Minimizing Scheduler

4.1 Notation

Let $\mathcal{W}$ denote the set of all WFGs to be scheduled for execution. For each $\boldsymbol{w} \in \mathcal{W}$ there is assumed to be a cost function, $F_{\boldsymbol{w}}(\tau_{\boldsymbol{w}})$, which maps a normalized measure of $\boldsymbol{w}$'s tardiness, $\tau_{\boldsymbol{w}}$, to a cost value. The total cost of the system, denoted by $F(\boldsymbol{\tau})$, is defined by summing the costs of all WFGs:

$$F(\boldsymbol{\tau}) = \sum_{\boldsymbol{w} \in \mathcal{W}} F_{\boldsymbol{w}}(\tau_{\boldsymbol{w}}), \tag{5}$$

where $\boldsymbol{\tau} = [\tau_{\boldsymbol{w}}]_{\boldsymbol{w} \in \mathcal{W}}$.

The normalized tardiness of WFG $\boldsymbol{w}$ is defined by the follow equation:

$$\tau_{\boldsymbol{w}} = \frac{f_{\boldsymbol{w}} - d_{\boldsymbol{w}}}{d_{\boldsymbol{w}} - b_{\boldsymbol{w}}}. \tag{6}$$

The numerator of the expression, $f_{\boldsymbol{w}} - d_{\boldsymbol{w}}$, represents the actual tardiness of $\boldsymbol{w}$. The denominator of the expression, $d_{\boldsymbol{w}} - b_{\boldsymbol{w}}$, represents the maximum desired amount of time allocated for executing $\boldsymbol{w}$, and is by definition positive. The numerator can be either positive or negative. Thus, $\tau_{\boldsymbol{w}} \leq 0$ indicates that $\boldsymbol{w}$ is not tardy and $\tau_{\boldsymbol{w}} > 0$ indicates $\boldsymbol{w}$ is tardy.

Because $\tau_{\boldsymbol{w}}$ is normalized, it is straightforward to compare the relative tardiness values of WFGs of different sizes and/or expected durations. For instance,

an actual tardiness of $f_{\boldsymbol{w}} - d_{\boldsymbol{w}} = 10$ seconds is relatively insignificant if the overall allocated duration is $d_{\boldsymbol{w}} - b_{\boldsymbol{w}} = 1$ hour, i.e., $\tau_{\boldsymbol{w}} = \frac{10}{3600} = 0.0028$. However, a tardiness of 10 seconds could be quite significant if the overall allocated duration is defined to be 40 seconds, i.e., $\tau_{\boldsymbol{w}} = \frac{10}{40} = 0.25$.

In order to derive an effective cost-minimizing scheduler, it is convenient to assume that the WFG functions $F_{\boldsymbol{w}}(\tau_{\boldsymbol{w}})$ are non-decreasing functions. This is a reasonable assumption in practice because a sensible SLA should not allow greater tardiness to be less costly than any lesser tardiness.

4.2 Cost-Minimizing Scheduling Algorithm (CMSA)

The function of CMSA is to decide which, if any, of the "ready" requests present in the scheduling pool should be assigned to a machine to begin execution. Scheduling decisions are implemented only at discrete points in time defined as *scheduling instances*. Two events can trigger a scheduling instance: (1) when a request finishes execution or (2) when a new WFG arrives in the scheduling pool. During the time period between two consecutive scheduling instances, the currently executing requests continue executing and the states of the requests in the scheduling pool do not change. Also, based on the machine model described in the previous section, the efficiency value, e, of each machine does not change during the time period between consecutive scheduling instances.

At each scheduling instance, and for each ready request in the scheduling pool, CMSA decides whether to start a request on a machine, based on the outcome of cost function analysis. Specifically, the scheduler estimates the cost associated with starting a ready request now (at the current scheduling instance) or holding the request in the pool until a future scheduling instance. Central to the algorithm's decision-making process is the ability to estimate the costs associated with competing scheduling options. A primary source of uncertainty in estimating a WFG's cost, $F_{\boldsymbol{w}}(\tau_{\boldsymbol{w}})$, is estimating the finish time, $f_{\boldsymbol{w}}$, of the WFG. Recall from Eq. 6 that $\tau_{\boldsymbol{w}}$ is directly proportional to $f_{\boldsymbol{w}}$.

Predicting the exact value of $f_{\boldsymbol{w}}$ (before $\boldsymbol{w}$ has finished execution) is generally not possible because all scheduling decisions, including those yet to be made, ultimately affect the values of $f_{\boldsymbol{w}}$ for all WFGs. As is apparent from Fig. 3, the issue of how to best estimate the finish time of even a single request is not obvious because the value of f_r depends on factors in addition to the request's start time s_r, including how the efficiency of the machine on which it is executing varies with time.

For the purposes of the present discussion, an estimate is assumed to be available for $\boldsymbol{w}$'s finish time at scheduling instance t_i, and this estimate is denoted by $\tilde{f}_{\boldsymbol{w}}(t_i)$. A description of the particular method used to calculate $\tilde{f}_{\boldsymbol{w}}(t_i)$ in the simulation studies is provided in Section 5.

Let $\mathcal{M}$ denote the set of machines and $M(t_i)$ denote the set of requests currently executing on machine $M \in \mathcal{M}$ at scheduling instance t_i. Let $R(t_i)$ denote the set of ready requests in the scheduling pool at scheduling instance t_i, and let $\boldsymbol{w}(r)$ denote the WFG associated with request r.

Basic Scheduling Decision: A basic decision made by the scheduling algorithm involves deciding whether to start executing a ready request at a current scheduling instance or to wait until a future scheduling instance. This basic decision assumes a candidate ready request and a candidate machine are specified.

> For ready request $r \in R(t_i)$ and machine $M \in \mathcal{M}$, determine whether it is less costly to start r on M at the current scheduling instance t_i or wait until a future scheduling instance $t_M > t_i$.

The value of t_M is defined to be the next scheduling instance generated by machine M due to the completion of one of M's executing requests. The value of t_M is itself dependant upon whether a particular ready request r^* is started at instance t_i. The formulas for the two possible values of t_M, denoted t_M^{wait} and t_M^{start}, are given by:

$$t_M^{\text{wait}} = t_i + \min_{r \in M} \left\{ \frac{c_r(t_i)}{U_r} \frac{1}{e^{\text{wait}}} \right\} \tag{7}$$

$$t_M^{\text{start}} = t_i + \min_{r \in M \cup \{r^*\}} \left\{ \frac{c_r(t_i)}{U_r} \frac{1}{e^{\text{start}}} \right\}, \tag{8}$$

where $e^{\text{wait}} = e(M(t_i), t_i)$ and $e^{\text{start}} = e(M(t_i) \cup \{r^*\}, t_i)$.

For convenience, define $\Delta t^{\text{wait}} = t_M^{\text{wait}} - t_i$ and $\Delta t^{\text{start}} = t_M^{\text{start}} - t_i$. The cost associated with waiting until t_M^{wait} to begin executing r^* on M is defined by:

$$\Gamma_{r^*,M}^{\text{wait}} = \Gamma_{\boldsymbol{w}(r^*)} \left(\frac{\tilde{f}_{\boldsymbol{w}(r^*)} + \Delta t^{\text{wait}} - d_{\boldsymbol{w}(r^*)}}{d_{\boldsymbol{w}(r^*)} - b_{\boldsymbol{w}(r^*)}} \right) + \sum_{r \in M} \Gamma_{\boldsymbol{w}(r)} \left(\frac{\tilde{f}_{\boldsymbol{w}(r)} - d_{\boldsymbol{w}(r)}}{d_{\boldsymbol{w}(r)} - b_{\boldsymbol{w}(r)}} \right). \tag{9}$$

The cost associated with starting r^* on M at time t_i is defined by:

$$\Gamma_{r^*,M}^{\text{start}} = \sum_{r \in M \cup \{r^*\}} \Gamma_{\boldsymbol{w}(r)} \left(\frac{\tilde{f}_{\boldsymbol{w}(r)} + \Delta t^{\text{start}} \left(\frac{1}{e^{\text{start}}} - \frac{1}{e^{\text{wait}}} \right) - d_{\boldsymbol{w}(r)}}{d_{\boldsymbol{w}(r)} - b_{\boldsymbol{w}(r)}} \right). \tag{10}$$

For each ready request $r \in R(t_i)$ and each machine $M \in \mathcal{M}$, the cost-minimizing algorithm computes the difference in costs $\Delta\Gamma_{r,M} = \Gamma_{r,M}^{\text{start}} - \Gamma_{r,M}^{\text{wait}}$. If $\Delta\Gamma_{r,M} > 0$ for all $r \in R(t_i)$ and for all $M \in \mathcal{M}$, then the scheduler will not start any request now (at scheduling instance t_i). However, if there exists one or more combinations of requests and machines for which $\Delta\Gamma_{r,M} \leq 0$, then the scheduler will start the request on the machine having the smallest starting penalty, defined as follows:

$$\Gamma_{r,M}^{\text{penalty}} = \Delta\Gamma_{r,M} + \Gamma_{\boldsymbol{w}(r)} \left(\frac{\tilde{f}_{\boldsymbol{w}(r)} + \Delta t^{\text{wait}} - d_{\boldsymbol{w}(r)}}{d_{\boldsymbol{w}(r)} - b_{\boldsymbol{w}(r)}} \right). \tag{11}$$

Fig. 5 provides the precise description of CMSA. For a given scheduling instance t_i, CMSA first performs computations for all combinations of ready requests and machines, refer to lines 3 through 11. After completing this phase of computation, CMSA then determines whether there exists a request that can be started on a machine. If the answer is no, then the algorithm exits, refer to lines 12 and 13. However, if the answer is yes, then the selected request is assigned to the selected machine (line 14), the selected request is removed from the set of ready requests (line 15), and the algorithm again performs computations for all combinations of ready requests and machines (line 16). The complexity associated with performing computations for all combinations of ready requests and machines is $O(|R(t_i)||\mathcal{M}|)$. Because it is possible that these computations may be performed up to $|R(t_i)|$ times, the worst case computational complexity of CMSA is $O(|R(t_i)|^2|\mathcal{M}|)$.

Note that if the system is highly loaded, then $|R(t_i)|$ will tend to be large. This is because a highly loaded system implies there are limited machine resources available to assign ready requests, thus ready requests will tend to accumulate in the scheduling pool. Because of this, it is likely that CMSA will exit soon under the highly loaded assumption, meaning that while $|R(t_i)|$ is large, the actual complexity of CMSA may be closer to $O(|R(t_i)||\mathcal{M}|)$ than $O(|R(t_i)|^2|\mathcal{M}|)$. On the other hand, if the system is lightly loaded, then $|R(t_i)|$ will tend to be small. This is because a lightly loaded system implies there are ample machine resources available to assign ready requests, thus ready requests will tend to be removed quickly from the scheduling pool. Thus, in the lightly loaded case, the complexity of CMSA tends to be characterized by $O(|R(t_i)|^2|\mathcal{M}|)$. However, because $|R(t_i)|$ is relatively small, the actual complexity for the lightly loaded case may be comparable to, or even less than, the complexity of CMSA under high loading.

```
1   for scheduling instance t_i
2     minPenalty ← ∞, r_min ← ∞, M_min ← ∞
3     for each r ∈ R(t_i)
4       for each M ∈ M
5         compute ΔF_{r,M} = F^start_{r,M} − F^wait_{r,M}
6         compute F^penalty_{r,M}
7         if ΔF_{r,M} ≤ 0
8           if F^penalty_{r,M} < minPenalty
9             minPenalty ← F^penalty_{r,M}
10            r_min ← r
11            M_min ← M
12    if minPenalty = ∞
13      exit
14    assign request r_min to machine M_min
15    R(t_i) ← R(t_i) − {r_min}
16    goto line 2
```

Fig. 5. Pseudocode for CMSA

5 Simulation Studies

CMSA is evaluated through simulation studies for a realistic scenario in which different types of WFGs are submitted to the cloud (refer to Fig. 1) by clients from three primary regions: Americas; Europe; and Asia. Furthermore, WFGs of three different types are submitted by clients: Batch, Webservice, and Interactive. Batch WFGs generally have a larger number of requests and requests with greater CPU and memory heap requirements compared to the other two WFG types. The Webservice WFGs generally have more requests than and requests with more requirements than Interactive WFGs.

In addition to differences in number and sizes of requests, the different WFG types are characterized by different arrival rates. The studies conducted were modeled from a typical 24 hour period observed in a live system. Webservice WFGs arrive uniformly over the 24 hours. Interactive WFGs arrive at constant rates only during three 8 hour periods that are partially overlapping. These periods represent interactive use by clients during normal working hours for the three client regions. The bulk of Batch WFGs arrive at hour seven and the arrival rate exponentially decays afterward. The arrival rates over a 24 hour period for the three types of WFGs are illustrated graphically in Fig. 6.

The parameter value ranges and distributions associated with the simulation studies are summarized in Table 2. The table defines parameters related to the structural characteristics for each type of WFG, which are all assumed to have a level of depth as the example in Fig. 2. Also provided in the table are CPU and heap memory characteristics of the requests associated with each WFG type. In all cases, a parallelization factor of two is used in determining a base deadline for each generated WFG; it defines the degree of parallelism assumed for executing parallel RCs from a common WFG. Once a base deadline is determined for a WFG, it is multiplied by the Deadline Factor (last row in the table) to define actual deadline for the WFG.

In making assignment decisions, the Scheduler can make use of computational and heap memory requirements assumed to be known and available for each request. Having access to such information is realistic in the assumed environment in which off-line profiling and/or historical logging can be performed to collect/estimate these data. Also associated with each WFG is a single timing deadline, and the Scheduler can also make use of WFG deadline requirements in making request scheduling decisions.

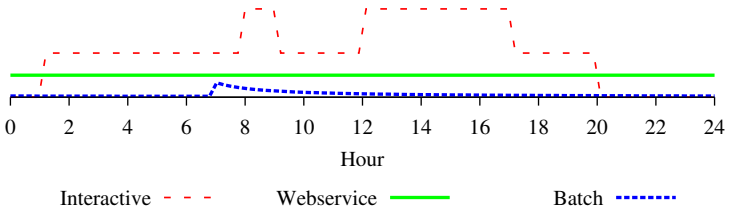


Fig. 6. Arrival rate of WFGs by type over a 24-hour period

Table 2. WFG parameter value ranges, [Min, Max], taken from uniform distributions for simulation studies

Parameter	Interactive WFG	Webservice WFG	Batch WFG
Compound Nodes	[1, 1]	[1, 3]	[3, 5]
Parallel RCs	[1, 2]	[2, 3]	[5, 20]
Requests in RCs	[5, 8]	[5, 8]	[3, 8]
Request Ideal Duration (secs), I_r	[1, 5]	[10, 30]	[50, 250]
Request CPU Utilization, U_r	[0.5, 1.0]	[0.5, 1.0]	[0.5, 1.0]
Request Heap Memory, H_r	[0.05, 0.1]	[0.05, 0.1]	[0.05, 0.15]
WFG Deadline Factor	[1.1, 1.2]	[1.3, 1.5]	[1.3, 1.5]

As described in Section 4, an estimate of each WFG's finish time, denoted as $\tilde{f}_{\boldsymbol{w}}(t_i)$, is necessary for the CMSA. The following formula is used to estimate WFG finish time in the simulation studies:

$$\tilde{f}_{\boldsymbol{w}}(t_i) = t_i + (t_i - s_{\boldsymbol{w}}) \left(\sum_{r \in \boldsymbol{w}} C_r - \sum_{\substack{r \in w \\ f_r < t_i}} C_r \right) \Bigg/ \left(\sum_{\substack{r \in w \\ f_r < t_i}} C_r \right) . \tag{12}$$

The CMSA is evaluated against a previously known algorithm, proportional least laxity first (PLLF) [14], which prioritizes scheduling requests with the least estimated proportional laxity, which is equivalent to the greatest normalized tardiness defined in Eq. 6. PLLF does not make use of any cost function, but only defines the order in which requests are considered for scheduling and relies on a separate policy to decide what machine to start the request on or when to forego scheduling ready requests. In the studies presented PLLF is combined with an algorithm that selects the machine based on the one that will have the largest values of e^{start}, which is defined as the efficiency that results if the request is started on that machine at the current time instance. PLLF elects to forego scheduling requests if all machines' values of e^{start} are below a prescribed threshold value.

Two sets of simulation studies were conducted, one with a sigmoid cost function and the other with a quadratic cost function. Fig. 7 shows the percentage of workflows whose normalized tardiness is at or below the given value of normalized tardiness. For example, only about 30% of workflows scheduled using PLLF had a normalized tardiness of zero or less (met their deadline or were early). In contrast, over 90% of the workflows scheduled by CMSA (using the sigmoid cost function) met their deadline. Also illustrated for reference are the sigmoid and quadratic cost functions. Fig. 8 shows the cumulative running cost of workflows by born time across the 24 hour simulated study period for both cost functions. Although PLLF does not explicitly use a cost function, it was evaluated using the same cost function used by CMSA. The cumulative running cost of both algorithms coincide during the zero to seven hour period, which represented a period when the system is lightly loaded. However, after this point the PLLF algorithm makes very different scheduling decisions than CMSA.

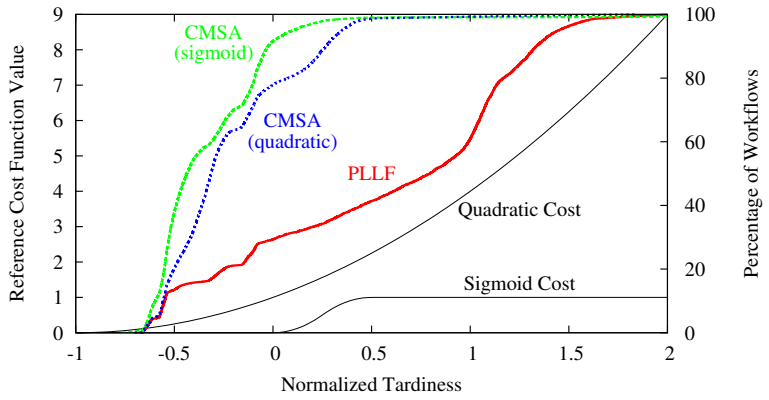
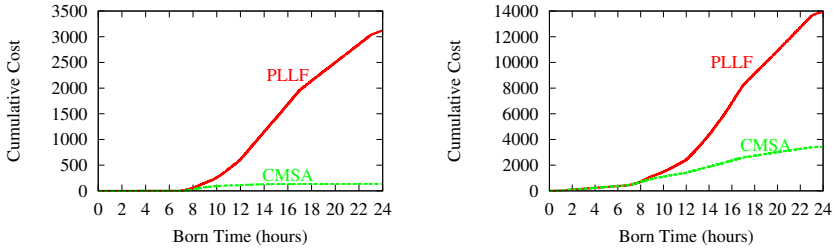


Fig. 7. Percentage of workflows as a function of normalized tardiness for PLLF and CMSA. Also shown are the reference sigmoid and quadratic cost functions.



(a) (b)

Fig. 8. The cumulative running cost of all WFGs by born time for PLLF and CMSA assuming: (a) sigmoid and (b) quadratic cost functions

Table 3. Summary of results of simulation studies

Measure	**Sigmoid**		**Quadratic**	
	PLLF	**CMSA**	**PLLF**	**CMSA**
Cumulative Cost	3,121.7	135.8	13,935.7	3,432.5
% Workflows Late	70.7	8.4	70.7	22.2
% Interactive WFGs Late	74.9	14.0	74.9	37.6
% Batch WFGS Late	96.6	41.5	96.6	57.6
% Webservice WFGs Late	64.3	0.1	64.3	2.3
Normalized Tardiness 95th percentile	1.45	0.10	1.45	0.35
Normalized Tardiness 99th percentile	1.73	0.61	1.73	0.75
Maximum Normalized Tardiness	2.09	5.10	2.09	1.85

Table 3 gives a quantitative summary of all results. From the table, the scheduling produced by PLLF is relatively closer to CMSA for the case of a quadratic cost function ($13,935.7/3,432.5 \approx 4 < 3,121.7/135.8 \approx 23$). This is because PLLF elects to work on WFGs that are estimated to be most tardy and CMSA ultimately does as well due to the unbounded increasing nature of the quadratic cost function.

In the case study using the sigmoid cost function the CMSA achieves lower normalized tardinesses for the vast majority of WFGs due to the fact that the sigmoid cost function limits the cost of WFGs with normalized tardiness values greater than 0.5. Refer to the table data for the normalized tardiness values of the 95th and 99th percentiles, as well as the maximum normalized tardiness for each policy.

6 Conclusions

A new cost-minimizing scheduling algorithm (CMSA) is introduced for scheduling requests of multi-level workflows of various types and degrees of complexity. The algorithm assumes a cost function is provided, and operates by making scheduling decisions in order to minimize the estimated value of cumulative cost. The performance of the new algorithm is evaluated through realistic simulation studies and compared to a previously best-known scheduling heuristic named PLLF, which is a priority-based scheduler that attempts to minimize maximum normalized tardiness. The simulation studies show that for both sigmoid and quadratic cost functions, CMSA results in maximum normalized tardiness values less than those for PLLF for over 99% of the workflows. Using the sigmoid cost function with CMSA, only about 8% of the workflows were tardy; in contrast, for the same scenario, over 70% of the workflows were tardy using the PLLF policy.

References

[1] Beltrán, M., Guzmán, A., Bosque, J.L.: A new cpu availability prediction model for time-shared systems. IEEE Transactions on Computers 57(7), 865–875 (2008)
[2] Zhang, Y., Sun, W., Inoguchi, Y.: Predicting running time of grid tasks on cpu load predictions. Proceedings of the 7th IEEE/ACM International Conference on Grid Computing, 286–292 (September 2006)
[3] Appel, A.W.: Garbage collection can be faster than stack allocation. Information Processing Letters 25(4), 275–279 (1987)
[4] Hertz, M.: Quantifying and Improving the Performance of Garbage Collection. Ph.D. Dissertation, University of Massachusetts, Amherst (2006)
[5] Hertz, M., Berger, E.D.: Quantifying the performance of garbage collection vs. explicit memory management. In: Proceedings of the Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2005) (October 2005)
[6] Jones, R., Lins, R.: Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley & Sons, New York (1996)

[7] Koide, H., Oie, Y.: A new task scheduling method for distributed programs that require memory management. Concurrency and Computation: Practice and Experience 18, 941–945 (2006)
[8] Dhakal, S., Hayat, M.M., Pezoa, J.E., Yang, C., Bader, D.A.: Dynamic load balancing in distributed systems in the presence of delays: A regeneration-theory approach. IEEE Transactions on Parallel & Distributed Systems 18(4), 485–497 (2007)
[9] Dyachuk, D., Deters, R.: Using sla context to ensure quality of service for composite services. IEEE Transactions on Computers 57(7), 865–875 (2008)
[10] Kim, J.K., Shivle, S., Siegel, H.J., Maciejewski, A.A., Braun, T., Schneider, M., Tideman, S., Chitta, R., Dilmaghani, R.B., Joshi, R., Kaul, A., Sharma, A., Sripada, S., Vangari, P., Yellampalli, S.S.: Dynamic mapping in a heterogeneous environment with tasks having priorities and multiple deadlines. In: 12th Heterogeneous Computing Workshop (HCW 2003), Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003) (April 2003)
[11] Oh, S.H., Yang, S.M.: A modified least-laxity-first scheduling algorithm for real-time tasks. In: Proceedings of the 5th International Workshop on Real-Time Computing Systems and Applications (RTCSA 1998), October 1998, pp. 31–36 (1998)
[12] Salmani, V., Naghibzadeh, M., Habibi, A., Deldari, H.: Quantitative comparison of job-level dynamic scheduling policies in parallel real-time systems. In: Proceedings TENCON, 2006 IEEE Region 10 Conference (November 2006)
[13] Feizabadi, Y., Back, G.: Garbage collection-aware utility accrual scheduling. Real-Time Systems 36(1-2), 3–22 (2007)
[14] Shrestha, H.K., Grounds, N., Madden, J., Martin, M., Antonio, J.K., Sachs, J., Zuech, J., Sanchez, C.: Scheduling workflows on a cluster of memory managed multicore machines. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2009 (July 2009)
[15] Dertouzos, M.L., Mok, A.K.-l.: Multiprocessor on-line scheduling of hard-real-time tasks. IEEE Transactions on Software Engineering 15(12), 1497–1506 (1989)