UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

FEEDBACK AND REQUIREMENT BIASING FOR ENHANCING ROBUSTNESS OF SCHEDULING ALGORITHMS FOR DISTRIBUTED SYSTEM PROCESSING

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

DOCTOR OF PHILOSOPHY

By

NICOLAS GENE GROUNDS
Norman, Oklahoma
2018

FEEDBACK AND REQUIREMENT BIASING FOR ENHANCING ROBUSTNESS OF SCHEDULING ALGORITHMS FOR DISTRIBUTED SYSTEM PROCESSING

A DISSERTATION APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY

Dr. John Antonio, Chair

Dr. Sridhar Radhakrishnan

Dr. S Lakshmivarahan

Dr. Sudarshan Dhall

Dr. Kash Barker

Acknowledgements

First, I wish to express my gratitude to my research advisor, Dr. John Antonio, for teaching me a great deal about research and about life. The years I have known and worked with him have been incredible and I owe him much more than this humble attempt at gratitude. He will forever be a mentor and close friend.

I also wish to thank the other members of my disseration research committee: Dr. Sridhar Radhakrishnan, Dr. S Lakshmivarahan, Dr. Sudarshan Dhall, and Dr. Kash Barker. Thank you for your service as committee members and for all the time and energy you put into reviewing my research materials and providing your feedback. Thank you also for the various courses which you've taught and I have taken over the years. From each of you I've learned a great deal.

I also wish to thank my employer and various managers throughout the past nine years who have allowed me the flexibility to pursue this research and my involvement in academics even though it at times interfered with my normal work activities. Thank you especially to Jeff Muehring, a friend and mentor as well as manager, who first allowed and encouraged me to pursue a doctoral degree.

Finally, thank you to my family who had to endure my absence or absence-mindedness especially at times when deadlines loomed and my research interested consumed so much of my mind, energy, and time.

Contents

Abstract

Scheduling tasks in a distributed system (e.g., cloud computing) in order to optimize an objective such as minimizing deadline misses has been a topic of research for decades. Such a problem is proven NP Complete even with perfect knowledge of tasks' requirements and their arrivals to the distributed system for processing. However, a realistic approach to distributed systems scheduling as part of a cloud computing service provider requires development of scheduling algorithms that can accomodate tasks arriving dynamically having requirements that are not accurately known.

In this dissertation, a framework is proposed for handling dynamic scheduling of tasks where scheduling algorithms decision-making is based on execution within a modeled system of modeled tasks with inaccurate requirements with respect to the actual tasks running on an actual system. Tasks are arranged into directed-acyclic graphs representing execution precedence constraints called workflows, which have a known deadline or time by which all contained tasks should complete processing. Various scheduling algorithms are evaluated and compared using simulation software (simulating both the model and actual systems) according to their ability to complete workflows relative to their deadline as well as their robustness to the amount of error in the model tasks' requirements. Simulation conditions are varied with respect to the amount of error in model tasks' requirements. Finally, feedback from the actual system to the model system regarding task processing completion and model error biasing techniques are evaluated and shown to be useful in enhancing the robustness of scheduling algorithms to model errors.

Chapter 1

Introduction

1.1 Overview

Scheduling or mapping tasks of heterogenous requirements to a collection of resources with the capabilities of processing the tasks in order to achieve an objective (e.g., minimizing makespan, which is the last completed task's completion time) is, even in its simplicity, an intractable problem, akin to the bin-packing problem [3]. Common extensions to this scheduling problem include dependency (i.e., precedence) constraints between some tasks and an objective of finishing some or all tasks by a specified deadline. Further extensions considered here include exploiting the opportunity for parallelism in task processing (which may reduce the performance of each individual task based on a non-linear processor efficiency model), and uncertainty (or error) in the estimations of task requirements. These extensions take what was already a "hard" problem and make it far harder. This is the problem to be addressed in this dissertation.

For the purpose of this research, computational work to be scheduled is divided into units known as tasks. Each task has certain requirements of computing resources (e.g., CPU cycles and memory allocation). A scheduling decision determines when a task's execution should begin and upon which machine (i.e., hardware or server) the task should be executed. Tasks are assigned to machines composed of the same resources for which tasks have requirements, but also have an efficiency function that governs how quickly a unit of work can be executed for a task, based on the amount of resources consumed by all tasks executing concurrently on that machine. For example, a task with substantial memory requirements can place more strain on the memory management subsystem of the machine and thus take longer to execute than a similar task with an equal amount of computational work to be done, but which requires less memory. Additionally, as machines' resources may be shared there may be multiple tasks executing concurrently on the machine, thereby raising the aggregate resource utilization and driving the machine efficiency even lower. A scheduling decision may attempt to increase productivity by increasing the number of tasks executing concurrently in order to make progress on each task equally as opposed to sequentially executing the tasks and finishing some tasks far sooner than others.

Some tasks may be associated with others via precedence constraints. These precedence relationships lend themselves to modeling the aggregate of all associated tasks in a directed, acyclic graph (DAG). These superstructures of all tasks related through precedence are termed a workflow. Workflows are representative of overall jobs that are to be executed as a collection of tasks which may be scheduled semi-independently (a task may not begin execution until all precedent tasks have completed execution). Figure 1.1 depicts a simple workflow in the form of a DAG. Note that this research makes no assumptions about the nature or structure of the precedence relationships between tasks in a workflow. For example, a workflow may be more than a single, linear chain of tasks (i.e.,

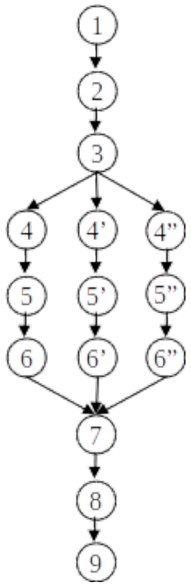

Figure 1.1: Example workflow depicted as a DAG of tasks.

more than one task may share a single, precedent task). This allows scheduling decisions that execute more than one task of the same workflow in parallel so long as all precedent tasks are completed. For example, from Figure 1.1 once task 3 has completed, tasks 4, $4'$, and $4''$ may each be scheduled to begin at the same or different times as neither of the number 4 tasks have a precedence constraint on another number 4 task.

Each workflow, as a representation of a job, has an associated deadline, or time by which the workflow (i.e., all its tasks) should be completed. One basic assumption in this research is that the computing capacity of all available machines is insufficient for processing the tasks of all workflows as they arrive to be executed in the overall system. If this were not the case and all workflows' tasks could be executed immediately upon arrival (whether as the first task of the workflow, or immediately after all precedent tasks have completed) then there would be no need for any decision-making about when to begin execution of a task, nor even the machine upon which to begin its execution. This creates a need for a schedule (or set of scheduling decisions) determining whether each task should begin as soon as ready or delayed in order to allow other, executing tasks to complete their execution first. The performance of a schedule is measured, ultimately, by whether or to what degree it completes workflows by their deadlines.

A scheduling algorithm is an algorithm that determines scheduling decisions for a start time of execution and machine upon which to execute each task associated with all workflows. This set of all scheduling decisions is also known as a schedule. A schedule thus is a static set of decisions, meaning a start time and machine assignment for each task. Thus, a static schedule is only suitable for a particular scenario of workflows, tasks, and a given set of machines; whereas a scheduling algorithm could be applied to different contexts in order to produce appropriate schedules. A scheduling algorithm that requires knowledge of all workflows and tasks ahead of executing any scheduling decision is known as a static scheduling algorithm. A dynamic algorithm, on the other hand, is able to make scheduling decisions given only a subset of information (e.g., only knowledge of workflows that have tasks executing, previously executed, or ready to begin executing, but no knowledge of workflows which may arrive in the future) [15]. A dynamic scheduling algorithm could be used "online" during the execution of a system of machines, workflows and their tasks. In contrast, a static scheduling algorithm is designed to operate "offline," with the resulting schedule being applied in the "online" environment.

1.2 Measures of Performance

Performance of a dynamic scheduling algorithm may be measured by its degree of achieving a desired outcome such as minimizing the number of workflows completed "late" (after their deadline). Another typical performance metric is to minimize the maximum lateness (or tardiness) of any workflow. Because different workflows may have different magnitudes of overall processing time, it is useful to normalize tardiness by the time difference (or other function) between the worflow's deadline and arrival time into the system.

Naturally, some algorithms may prioritize optimizing one performance metric over another, though it often comes at the expense of sacrificing performance measured by other metrics. For example, in a system overloaded and incapable of finishing all workflows before their deadline one scheduling algorithm may prefer to maximize the number of late workflows by "sacrificing" one workflow and not processing it at all until after all other workflows are completed. In this way all but one worflow may be completed on time, but the sacrificed workflow's tardiness is substantial. Another algorithm's behavior may be to minimize the maximum tardiness across all workflows and thus to prevent any one worflow from being very late, it incrementally processes subsets of all workflows, making them all late by a small amount.

Achieving optimal performance may be prevented for several reasons. First, the granularity of division of work among various tasks of a workflow may be such that no scheduling decisions can, for example, work at equal rates on all workflows in order to achieve equally-minimalized tardiness across all workflows. Furthermore different required resources of tasks in different workflows such as simply CPU vs. memory requirements may cause the efficiency of processing for different tasks to be unequal, or in the case of resources which may be unshareable or require synchronized access could cause bottlenecks in processing due to sequentialized access. Finally, it may be that the information about workflows and their tasks available to the scheduling algorithm may have error, which is discussed more in the following section.

1.3 Sources of Error

When a scheduling algorithm makes scheduling decisions of which workflow's task(s) to begin processing and on which machine it may require knowledge both about the workflow such as how many remaining tasks must be processed in order to complete processing of the workflow and characteristics about the task(s) of the workflow such as their resource requirements in order to predict their processing time. In many systems it may be impractical to have such knowledge available a priori. For example, in systems where the exact nature of the tasks to be processed as a workflow are in part based on the input data being processed it may not be possible to know a priori how many or what tasks will make up the workflow. Furthermore even the resource requirements of known (e.g., non-conditional or required) tasks may be based in part on values of input data that are not known at runtime.

In addition, for systems in which multiple tasks may be executed concurrently on shared resources (e.g., such as time-sharing a CPU resource) the cumulative effect of processing more than one task may be a source of uncertainty [4]. Where two tasks each require a fixed number of CPU cycles for processing and may in isolation have predictable processing time, the effect of allowing both tasks to execute concurrently may not be easily predictable because factors such as the cost and periodicity of context switching may not be known. Where tasks require synchronized (i.e., exclusive) access to a shared resource, they could cause delays in processing other tasks concurrently executing.

Therefore, the information provided to scheduling algorithms about workflows and the nature of each of workflow's tasks may be conceptualized as a "model" of the actual workflow and tasks in which there is error that causes the model to deviate from its "actual" counterpart. The distribution of this error may be estimated in some cases. For example, based on past observation of the running system, the distribution of tasks' resource requirements or workflow structure may be bounded (or even estimated as a random number distribution with an estimated distribution function such as uniform or Normal distribution).

1.4 Measures of Robustness

In the presense of error in the "model" a scheduling algorithm's robustness to that error may be measured in two ways. First, one may measure how many scheduling decisions are made differently in a scenario with error vs. a scenario without error. In effect this is a measurement of how different of a schedule is produced by the algorithm when error is present as opposed to having perfect information. This measurement while conceptually interesting may be of little practical use because even though a fundamentally different set of scheduling decisions is made, the algorithm may still achieve the same or similar outcome of performance.

The second measurement of robustness is to measure the effect of model error on the quality of scheduling decisions produced by the algorithm with respect to the chosen performance metric(s). In this case one or more performance metrics are measured and compared between the scenario with model error present vs. a model with no error.

The purpose of this research is to explore the performance characteristics of several dynamic scheduling algorithms, and the robustness of these algorithms with respect to their performance in the presence of error or uncertainty in the information available to the algorithm. One of the scheduling algorithms evaluated is also introduced in Chapter 4.

1.5 Main Results and Outline of Dissertation

The main results of this research are a demonstration that feedback from the "actual" system to the model of task completions is critical in achieving robustness to even small errors in the model. Without such feedback of all tasks' completions some amount of robustness to certain levels of error can be achieved by biasing the model task requirements to prevent model error from underestimating task requirements (and thus modeling tasks finishing earlier than the "actual" system). These results are based on simulation studies of a distributed system scheduling framework presented in Chapter 3 and presented for scheduling algorithms developed in previous research as well as a novel scheduling algorithm presented in Chapter 4.

In Chapter 5 the framework for measuring and evaluating robustness as well as the approach to using task completion feedback and biasing strategies to increase robustness are described. Results of simulation studies are presented and discussed in Chapter 6. Chapter 7 contains final conclusions of the research presented.

Chapter 2

Previous Work & Literature Review

2.1 Overview

Research into various methods and algorithms for scheduling tasks in computing environments has been well-established for several decades. Early research focused on task scheduling in constrained environments where machines (i.e., resources for executing tasks) were capabable of executing only a single task at a time (e.g., scheduling tasks for processing on embedded systems [3]). More recently research has focused on "cloud computing" in which such a restriction is removed and systems are presumed to be both multi-tenant (i.e., many clients submitting work loads to the system for processing) and capable of processing multiple tasks concurrently. In this chapter a brief review of outside literature is provided as well as a summary of past research by this author related directly to the research presented in later chapters.

Early research into task scheduling in computing environments focused on simple tasks (not workflows of tasks) [6, 25]. This and similar research also focused on operating system-level scheduling for a single machine or embedded system with very limited processing resources (e.g., CPUs) that support only a single task executing at a given time [27]. Even under these conditions scheduling to achieve optimal task completions is proven NP-hard [3, 27]. Even in cases where tasks have interdependence or predecessor constraints such that one task must be completely executed before another can begin work such as [17] still only considered problems where processors can execute a single task at a time.

As task execution times (or resource utilization) increasingly became viewed as stochastic, evaluating scheduling "robustness" to such uncertainty became a more studied topic. In work such as [5] this robustness was defined as whether a static schedule (i.e., the set of scheduling decisions for which tasks to execute on which nodes of a heterogeneous environment of computing systems, produced by various scheduling algorithms) still achieved a similar makespan (time at which the last task was finished executing) despite task execution time uncertainty. In [22] robustness of dynamic scheduling is measured with respect to the amount of tasks completing ahead of their deadline.

In later years as distributed systems became more prolific task scheduling research naturally shifted to consider these distributed systems. Research also began focusing on the performance and impact of scheduling multiple tasks to a single distributed system node in order to achieve better resource utilization and meet more demanding scheduling constraints (e.g., deadlines) [8].

Similarly, as task scheduling research has evolved over the years so too has the degree of complexity of scheduling algorithms. Early scheduling algorithms were simple and considered only innate properties of tasks or workflows such as First-Come, First-Served (FCFS), which schedules based on which task or workflow arrived at the system for scheduling earliest. Another is the Earliest Deadline First (EDF) heuristic, which prioritizes tasks based on which has a deadline (or belongs to a workflow with a deadline) that occurs soonest [3, 24]. More advanced heurstics using techniques such as particle swarms [21], generic algorithms [30], and ant colony optimization [16] have also been applied to task scheduling problems. Such heuristics for scheduling can be used to build a schedule (a complete set of scheduling decisions) offline if all tasks'/workflows' arrival for scheduling is known a priori. This kind of offline scheduling algorithm is known as a static algorithm.

Later research into scheduling algorithms began to consider more complex properties of tasks such as a prediction of when tasks may complete [29], which ultimately rely on further heuristics to predict arrival of tasks (if not known a priori), their resource requirements, and even the impact of future scheduling decisions. Such algorithms are by nature dynamic [15] and rely on computations and predictions based on the current state of the system in order to make scheduling decisions. Many such algorithms focus on minimizing some notion of cost associated with either the inefficiencies of poor scheduling decisions, or the cost to operate system resources used in the execution of tasks [28], or the cost of not meeting scheduling objectives such as completing tasks or workflows by their deadline (e.g. failing to meet service-level agreements, SLAs, under contractual obligation with the customer submitting the task/workflow) [1, 7].

2.2 Cloud Computing

Largely within the past ten years research into scheduling of tasks has shifted in both terminology and focus into a computing paradigm known as "cloud computing." In it computing resources are abstracted away from the application software (typically via technologies known as virtual machines) and as such the computing resources are transformed into a kind of public utility that can be shared among customers whose work loads may be static or change over time. One of the primary concerns of such systems [2] is around performance of customer workflows (i.e., completion time relative to "hard" or "soft" deadlines) in such a dynamic system, which is complicated by the dynamic nature of having multiple customers submitting work loads to the system having unpredictable requirements and timing. Thus resource utilization and prioritization of workflows and tasks must be a primary feature of any algorithm or system responsible for scheduling tasks for execution in cloud computing machines.

Another concern within a "cloud computing" paradigm revolves around the monetary cost of operating the system, which is passed on as the cost of customers degree of system use (i.e. the amount of computing resources utilized by a customer's workflow of tasks). Thus many scheduling algorithms within "cloud computing" environments have been augmented or designed to account for budgetary constraints in addition to traditional objectives such as completing work before a deadline [26, 28].

In [1] a summary of scheduling algorithms and approaches are reviewed and organized into various taxonomies based on their focus on domain for scheduling, measurement of the quality of service (i.e., metric used to measure scheduling outcome fitness), and features of the tasks and system resources used to make scheduling decisions (e.g., resource utilization and/or allocation, task estimation, load balancing).

2.3 Previous Research

In this section, a summary of prior research by the author is given as it relates to research presented in future chapters.

Beginning in [24] scheduling algorithm research focused on measurements of scheduling algorithm performance such as dealine miss percentage and maximum, normalized tardiness (how much earlier or later than the deadline that the workflow was completed, normalized by the workflow's size and deadline tightness). In this work heuristics-based scheduling algorithms that simply prioritized which tasks should be executed next were considered (e.g., FCFS, EDF). A newly proposed Proportional Least Laxity First (PLLF) algorithm which prioritized tasks based on projected finish times and normalized tardiness was shown to achieve better performance relative to the other scheduling algorithms investigated.

In [11] the Cost-Minimization Scheduling Algorithm (CMSA) detailed later in Chapter 4 was first proposed. It's performance was benchmarked against algorithms from [24] and shown to be superior. It also features the ability to customize it's behavior in terms of desired outcome performance metrics based on a chosen cost function that maps a workflow's predicted or final-outcome normalized tardiness to a cost. Two such cost functions show how the algorithm can be tailored to achieve fewer missed deadlines or reduced maximum (among all workflows) normalized tardiness.

The most recent research has focused on use of a model of distributed system resources and the effect of error (or innaccuracies) in a model with respect to the "actual" system and how scheduling algorithms are affected by such modeling error. In [10] it is shown that feedback from the "actual" system to correct certain aspects of model error can be critical to scheduling performance for all studied algorithms.

The machine model used in previous research as the combination of CPUs and memory as resources capable of supporting concurrently executing tasks (a relaxing of the “one task at a time” assumption, which separates this research from most other scheduling research) was in part validated in [12]. In it the predictability of CPU performance of executing various combinations of tasks concurrently was measured, which validated the modeled performance used in the simulation software developed and employed throughout this and previous research [24, 11, 10] and also described in Chapter 3.

Chapter 3

Simulation Software

3.1 Overview

For all numeric results presented herein as with previous research [24, 11, 19, 18, 10] the same scheduling simulation software primarily developed by the author was used. This software is available as open-source software [9]. It constitutes about 17,000 lines of Java code. This chapter will describe this simulator software's capabilities and features.

At its heart the simulation software represents a set of tasks where each task has a finite set of resource requirements and is associated with a larger "job" represented as a directed acyclic graph (DAG) of tasks known as as workflow. Furthermore, the simulator represents a cluster (grid, or cloud) of machines each with finite resource capacities. Figure 3.1 depicts a basic UML Class diagram of these main components of the Simulator code base (many utility, helper, and other classes that aid proper object-oriented design of the software are omitted for brevity).

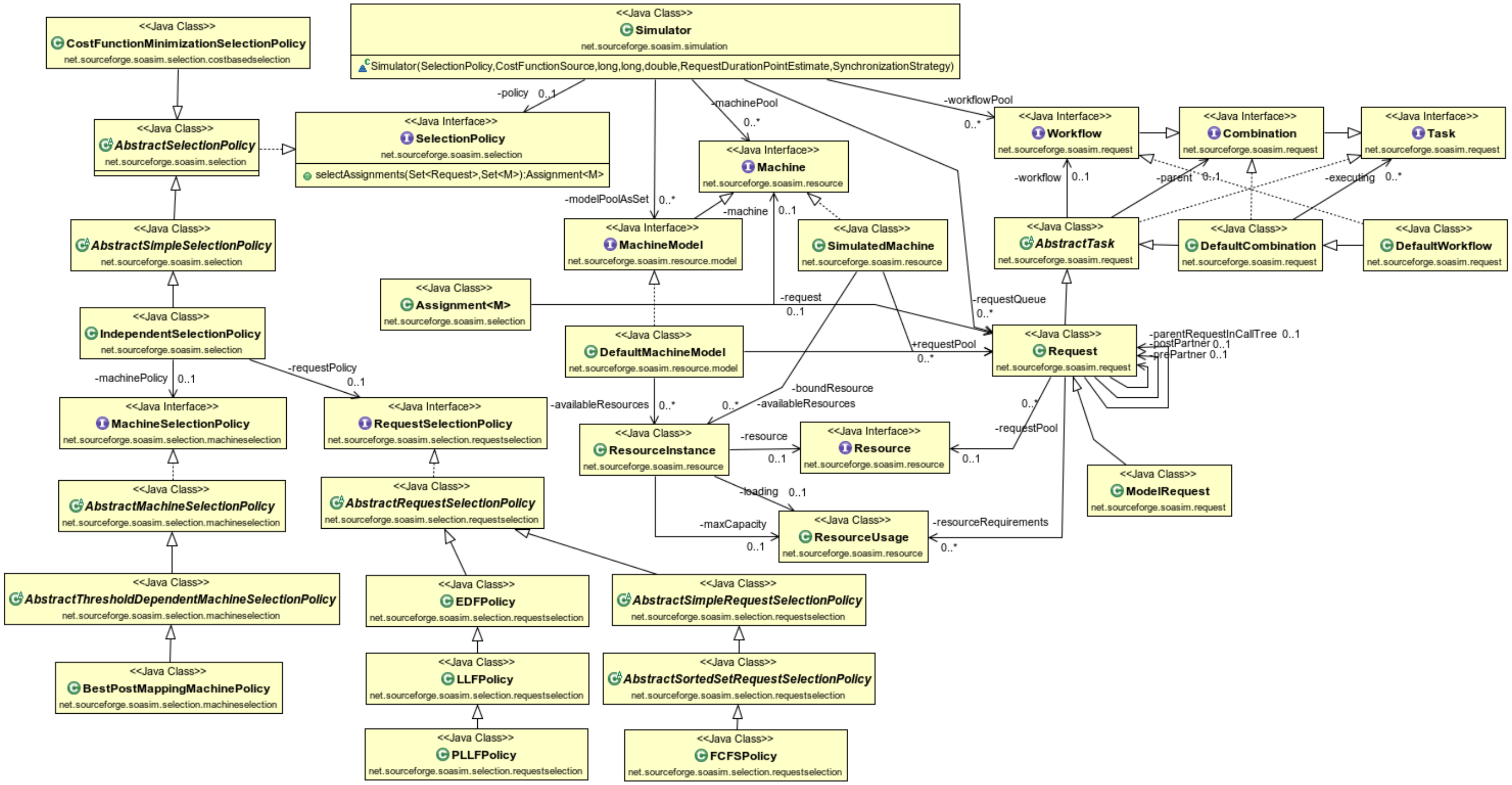
<<Java Class>>
Simulator
net.sourceforge.soasim.simulation
Simulator(SelectionPolicy,CostFunctionSource,long,long,double,RequestDurationPointEstimate,SynchronizationStrategy)
<<Java Class>>
CostFunctionMinimizationSelectionPolicy
net.sourceforge.soasim.selection.costbasedselection
<<Java Class>>
AbstractSelectionPolicy
net.sourceforge.soasim.selection
<<Java Interface>>
SelectionPolicy
net.sourceforge.soasim.selection
selectAssignments(Set<Request>,Set<M>):Assignment<M>
<<Java Class>>
AbstractSimpleSelectionPolicy
net.sourceforge.soasim.selection
<<Java Class>>
IndependentSelectionPolicy
net.sourceforge.soasim.selection
<<Java Interface>>
MachineSelectionPolicy
net.sourceforge.soasim.selection.machineselection
<<Java Class>>
AbstractMachineSelectionPolicy
net.sourceforge.soasim.selection.machineselection
<<Java Class>>
AbstractThresholdDependentMachineSelectionPolicy
net.sourceforge.soasim.selection.machineselection
<<Java Class>>
BestPostMappingMachinePolicy
net.sourceforge.soasim.selection.machineselection
<<Java Class>>
Assignment<M>
net.sourceforge.soasim.selection
<<Java Interface>>
RequestSelectionPolicy
net.sourceforge.soasim.selection.requestselection
<<Java Class>>
AbstractRequestSelectionPolicy
net.sourceforge.soasim.selection.requestselection
<<Java Class>>
EDFPolicy
net.sourceforge.soasim.selection.requestselection
<<Java Class>>
LLFPolicy
net.sourceforge.soasim.selection.requestselection
<<Java Class>>
PLLFPolicy
net.sourceforge.soasim.selection.requestselection
<<Java Class>>
AbstractSimpleRequestSelectionPolicy
net.sourceforge.soasim.selection.requestselection
<<Java Class>>
AbstractSortedSetRequestSelectionPolicy
net.sourceforge.soasim.selection.requestselection
<<Java Class>>
FCFSPolicy
net.sourceforge.soasim.selection.requestselection
<<Java Interface>>
Machine
net.sourceforge.soasim.resource
<<Java Interface>>
MachineModel
net.sourceforge.soasim.resource.model
<<Java Class>>
SimulatedMachine
net.sourceforge.soasim.resource
<<Java Class>>
DefaultMachineModel
net.sourceforge.soasim.resource.model
<<Java Class>>
ResourceInstance
net.sourceforge.soasim.resource
<<Java Interface>>
Resource
net.sourceforge.soasim.resource
<<Java Class>>
ResourceUsage
net.sourceforge.soasim.resource
<<Java Interface>>
Workflow
net.sourceforge.soasim.request
<<Java Interface>>
Combination
net.sourceforge.soasim.request
<<Java Interface>>
Task
net.sourceforge.soasim.request
<<Java Class>>
AbstractTask
net.sourceforge.soasim.request
<<Java Class>>
DefaultCombination
net.sourceforge.soasim.request
<<Java Class>>
DefaultWorkflow
net.sourceforge.soasim.request
<<Java Class>>
Request
net.sourceforge.soasim.request
<<Java Class>>
ModelRequest
net.sourceforge.soasim.request
-policy 0..1
-machinePool 0..*
-workflowPool 0..*
-modelPoolAsSet 0..*
-machine 0..1
-workflow 0..1
-parent 0..1
-executing 0..*
-request 0..1
-requestQueue 0..*
+requestPool 0..*
-parentRequestInCallTree 0..1
-postPartner 0..1
-prePartner 0..1
-machinePolicy 0..1
-requestPolicy 0..1
-availableResources 0..*
-boundResource 0..*
-availableResources
-resource 0..1
-requestPool 0..* 0..1
-loading 0..1
-maxCapacity 0..1
-resourceRequirements 0..*

Figure 3.1: UML Class diagram of the main component classes of the Simulator software beginning with the Simulator class which references a SelectionPolicy (scheduling algorithm), collection of Machines and ModelMachines (the actual and model platforms for a distributed system), queue of Requests (the ready tasks which may be scheduled), and a pool of Workflows which have arrived at the Simulator at a given point in time of the simulation.

The purpose of the simulator is to determine which queued tasks to begin executing on which machine and at what time. For this research the primary component of the software is the representation of a scheduling algorithm which, given the queue of tasks and a list of machines, will determine which tasks to immediately begin executing and on which machine. The scheduling algorithm is repeatedly invoked until either no tasks are left in the queue or the scheduling algorithm indicates it wishes to begin execution of no tasks.

The simulation is then driven forward by computing the next time in the simulated system when either a task completes execution or a new workflow arrives and its first task is enqueued for scheduling. The simulator then jumps to that future 'simulated' time, computes and effects the changes to executing tasks of how much 'work' was accomplished, adds any new workflows' ready tasks to the queue and allows the scheduling algorithm to once again determine which, if any, queued tasks should begin execution. This framework for event-driven simulation assumes that the "state" of the system that all scheduling algorithm use to determine whether to begin execution on any queued task consists only of the queue of ready tasks and the cumulative resource load of each machine (i.e. the cumulative effect of all currently executing tasks on a machine). Thus once a scheduling algorithm determines not to be begin executing any of the queued tasks, their is no need to invoke the algorithm again until either the queue of tasks is modified by a newly arriving workflow or an executing task completes.

The following sections describe in more detail various sub-components of the simulation software. Section 3.2 describes how workflows are modeled as an DAG of tasks. Section 3.3 describes how the arrival of new workflows occurs. Section 3.4 covers the model of resources and machines' resource capacities. Section 3.5 describes how a separate model for tasks and machines is used in the simulator.

Finally, Section 3.6 presents both how scheduling algorithms are defined and the four basic scheduling algorithms used in the simulation results in Chapter 6.

3.2 Workflow Model

A task is an individual, schedulable block of work. Tasks are defined by their resource requirements (i.e., amount of CPU utilization during each CPU clock cycle, number of CPU cycles required to complete the task's work, and amount of memory allocated). Tasks also belong to a workflow, which is a DAG of tasks. Each workflow is endowed with a deadline. Workflows represent the collection of work (i.e., tasks) necessary to accomplish a user's desired outcome or output in the distributed system.

As defined in [24] workflows are made up of various chains of tasks. Within a chain of tasks, each tasks must complete execution before the next task of the chain is ready for scheduling (i.e., placed in the queue of ready-to-execute tasks). Each workflow begins with a single chain of tasks (the length of the chain and the requirements of the tasks vary between various simulated types of workflows). Workflows also end with a chain of tasks. In between these chains are alternating blocks of parallel chains and single chains. Where task chains are executable in parallel the corresponding tasks are assumed to have equal resource requirements. This is used to represent parallel blocks of execution which are performing the same function (using the same amount of resources) on varying data input values.

Figure 3.2 (taken from [24]) illustrates how tasks are combined into chains and chains into sequential and parallel blocks to make up the DAG of a workflow.

For related research [18, 19] tasks can also be associated with one another as in a call tree. For example, one task, T_1, may complete a portion of its work but

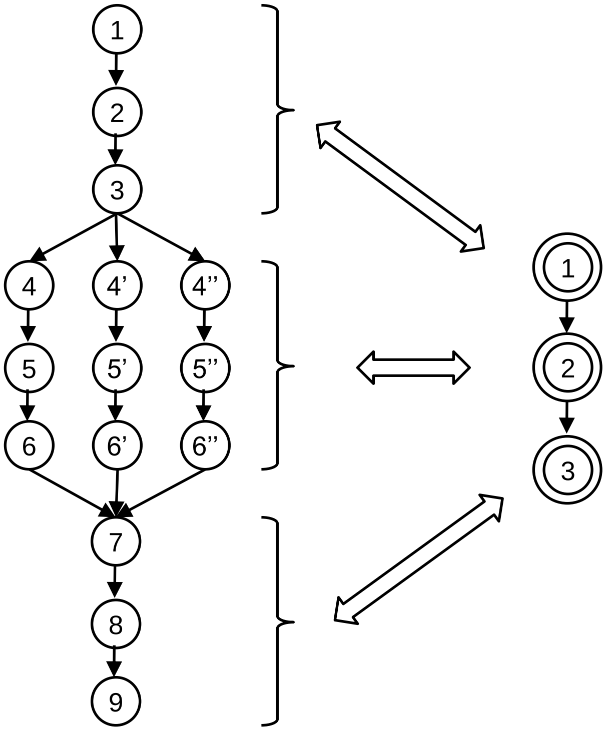

Figure 3.2: Figure 2 from [24] illustrating task chains (on the left) in parallel and sequential blocks (represented on the right)

then invoke another service of the distributed system which becomes a new task, T_2, to be scheduled for execution on any available machine. The execution of T_1 may synchronously block waiting for the result of T_2 during which time task T_1 may still consume some resource(s) (e.g., memory) though no longer consuming others (e.g., CPU). When T_2 completes, T_1 will resume execution immediately (i.e. without scheduling and on the same machine where it previous executed). This invocation concept can be nested resulting in a tree-structure known as a call tree. For the purpose of the simulator software call trees are 'unrolled' into chains of tasks where some tasks (i.e., the subsequent execution of a task after an invocation has been made, executed, and results returned) are marked such that they are not placed in the task queue for scheduling; instead they resume on the same machine where the work executed prior to the invocation. In the research results presented in Chapter 6, call trees are not present or used. In [18, 19] this capability of the Simulator was used to study detection and mitigation strategies for deadlock; a highly undesirable state in which all executing tasks are blocked waiting on invocations whose tasks cannot be scheduled because not enough resources are available in the distributed system.

3.3 Workflow Generation

The primary inputs into the Simulator are a configuration file describing the various types of workflows with their corresponding task resource requirements and a configuration file describing the number of machines and their resource capacities. The former is described further here and the latter in Section 3.4.

For convenience in parameterization, configuration of workflows can be divided into any number of "types" of workflows. Each type describes how individual workflows may be generated (number of tasks, chains, and blocks) and the simulated time at which they first arrive to be scheduled in the Simulator. For example, one workflow type may be for relatively small workflows of few tasks which arrive relatively often whereas another type may be for workflows consisting of a relatively large number of tasks, which arrive less frequently.

The arrival rate of each configured workflow type is specified as a parameterized mathematical function (or combination of functions). The functions implemented and provided in the Simulator software include one based on a Poisson distribution, an exponentially decaying function, a piecewise linear function, a simple predetermined arrival time list function, and a constant rate function. Additionally a combination function can be used to "overlay" multiple instances of the other functions as a sort of union of workflow arrivals from all those functions. At each time instant indicated by this arrival rate function a new, independent workflow is generated and its first task added to the Simulator's task queue.

The parameterization of the number of tasks for a generated workflow is specified as a minimum and maximum bounds (for a uniform distribution) for the various "abstraction levels" of the workflow. The last level specifies how many task objects will be generated in a single chain. The next-to-last level specifies how many chains may be combined in parallel blocks. A separate configuration parameter indicates whether two blocks of parallel chains may occur back-to-back as opposed to every block of parallel chains being preceeded and followed by a block consisting of a single chain. The next level specified how many blocks will be generated. Although the Simulator allows nesting more levels than these three, all past and current research has found no need for including levels beyond three.

Finally, the specification of individual tasks' resource requirements is specified in an XML configuration file where various task templates are specified. Each template parameterizes a random number distribution, the first for the work requirement (i.e., how many CPU cycles must be executed for the task to complete execution), and subsequently for the usage of each resource (e.g., memory, CPU). In past and current research only uniform distributions have been used for each resource requirement for all tasks (however, non-uniform distributiosn are assumed later in Chapter 6 when modeling errors in assumed resource requirements). As used in [18] (but not in this reasearch), the simulator also supports a nested set of tasks representing the invocation(s) made in a call tree. For each generated task either a random template is chosen, or by configuration the templates may be used in a round-robin fashion. An example XML configuration of a single task template (with no nested invocation tasks) is given in Figure 3.3

The work requirement of a task specifies how many units of the resource (CPU) must be executed in order to complete execution of the task. The CPU resource usage of a task specifies how many units of work the task completes on an ideal CPU in a unit of time. It is assumed the remaining time is spent in use of another, unspecified resource (e.g., input/output) when the CPU usage factor is less than 1.0. If two tasks were simultaneously executing on the same machine of a single CPU, and their cumulative CPU usage was less than or equal to 1.0, the Simulator assumes their ‘time-sharing’ of the CPU is ideal and both tasks’ rate of work is no different than if they were executing in isolation on the machine. More details on how the efficiency of CPU and memory resources are modeled is discussed in the following subsection.

```
<CallTrees>
  <CallTree>
    <Request>
      <Requirement>
        <Resource>CPU</Resource>
        <distribution name="uniform">
          <minValue>1.0</minValue>
          <maxValue>2.5</maxValue>
        </distribution>
      </Requirement>
      <Usages>
        <Usage>
          <Resource>Memory</Resource>
          <distribution name="uniform">
            <minValue>0.05</minValue>
            <maxValue>0.15</maxValue>
          </distribution>
        </Usage>
        <Usage>
          <Resource>CPU</Resource>
          <distribution name="uniform">
            <minValue>0.5</minValue>
            <maxValue>1.0</maxValue>
          </distribution>
        </Usage>
      </Usages>
    </Request>
  </CallTree>
</CallTrees>
```

Figure 3.3: Sample XML configuration of a single request template

3.4 Machine and Resource Representation

Other than the configuration of tasks and workflows as described in Section 3.3, the other primary input to the Simulator software is the parameterization of resources and machines. This occurs in a single configuration file that specifies the number and names of the resources that will be located on each machine, and the number of machines and each machine's capacity for those resources.

Generally, resources are assumed to be indepedent across machines. This is the case in past and current research where two resources, CPU and Memory, were simulated. However, the Simulator also supports configuration of "global" or shared resources that can be used to represent shared or centralized resources of a distributed system such as a relational database or distributed cache.

The number of machines is configurable. For each machine simulated an identification number is configured along with the capacity and function for computing efficiency for each of the independent (non-global) resources. In past and current research the simulated distributed system was compose of 16 machines each with identical capacities: 4 CPUs and a normalized memory capacity of 1. For the CPU resource an ideal efficiency function is used (i.e., if the cumulative usage of executing tasks was less than or equal to the capacity, 4, the efficiency is 100%, if usage was 8 then efficiency was 50%, usage of 12 was 33% efficient, and so on). Equation 3.1 formally defines this ideal CPU efficiency, e_{cpu}, as a function of cumulative CPU load of all executing tasks, ℓ_{cpu}, and CPU capacity, C.

$$e_{cpu} = \min\left(1, \frac{C}{\ell_{cpu}}\right) \tag{3.1}$$

For the memory resource, the function to compute efficieny iss non-linear and first presented in [24]. The memory efficiency function used assumed memory was managed and that increasing usage had a negligible effect at low levels of usage but efficience decreases dramatically for even small increases at high usage [13, 14]. Equation 3.2 defines the memory resource efficiency, e_{memory}, given cumulative memory load of all executing tasks, ℓ_{memory}.

$$e_{memory} = \frac{K}{K + \frac{1}{\frac{1}{\ell_{memory}} - 1}} \tag{3.2}$$

The ideal efficiency function used to model CPU resource efficiency is illustrated in Figure 3.4, originally presented in [24]. Figure 3.5, also from [24], illustrates the efficiency function used for the Memory resource.

The overall efficiency of a machine, e, is thus computed as the product of both CPU (e_{cpu}) and Memory (e_{memory}) resource efficiencies, as in Equation 3.3. This efficiency value, e, is then multiplied by the CPU utilization requirement of a task to compute how much actual CPU work is accomplished on that task per unit of time.

$$e = e_{cpu} e_{memory} \tag{3.3}$$

3.5 Model Subsystem

In the Simulator there are two distinct incarnations of the resources, machines, and workflows. The first is referred to as the actual platform; the second is the model platform. The model platform is used by the scheduler component for making scheduling decisions; the actual platform is used for measuring outcomes and performance of those scheduling decisions. The purpose of distinct platforms

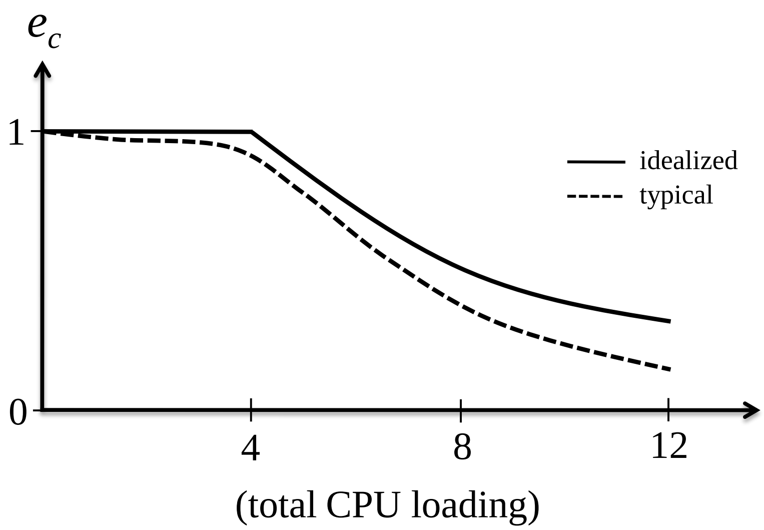

Figure 3.4: From [24] illustrating CPU resource's ideal efficiency function used by Simulator.

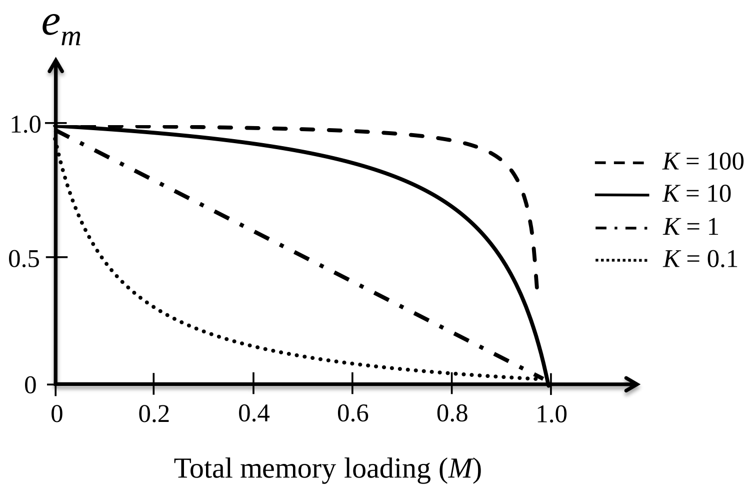

Figure 3.5: From [24] illustrating Memory resource's non-linear efficiency function used by Simulator.

within the Simulator is so that scheduling algorithms may be developed which may use a model of the platform for which they are scheduling tasks for execution which differs from the actual platform on which those scheduling decisions are realized. These differences may be intentional in that the model may be a simplified form of the actual platform to decrease the complexity requirement of the scheduling algorithm. The differences may also be incidental as in the model may have known errors or deviations due to the actual platform being too complex to model perfectly or having uncertainty due to external factors not related to the scheduler or executing tasks.

For the numeric results presented in Chapter 6 the performance of scheduling algorithms is measured on the actual platform for scenarios where the model platform has deliberate error terms applied to some or all resource requirements of all tasks, in order to measure the robustness of algorithms' ability to achieve similar performance in the presence of such modeling error vs. an error-free model.

3.6 Scheduling Algorithms

The heart of the Simulator is the scheduler component, which utilizes a scheduling algorithm to decide which ready tasks to begin executing and on which machine. As input scheduling algorithms are given the set of ready tasks and the set of machines from the model platform. Scheduling algorithms may return zero or more pairings of task-and-machine representing the decision to begin execution of task(s) on the machine(s). The scheduler component repeatedly invokes the scheduling algorithm until either the set of ready tasks is empty or the algorithm returns zero pairings.

There are four foundational scheduling algorithms implemented in the Simulator software. All four of these algorithms differ only in how they select which ready task is the highest priority task to begin executing next. Each of these four algorithms must also be paired with a secondary algorithm component to decide which machine to execute the highest priority task on, or whether no machine is either capable or all machines are too highly loaded to be desirable to begin execution of the task. In [24] some of these machine selection policies are discussed and ultimately the best-performing is one which selects the least loaded machine which would be above a static threshold for machine efficiency assuming the task were to begin execution immediately. The threshold represents a minimum preferred machine efficiency throughout the simulation scenario and prevents overloading machines during times when the queue of ready tasks is large.

The first of the four foundational scheduling algorithms is first-come-first-serve (FCFS), which selects tasks purely based on their time to become ready. This algorithm effectively renders the ready task queue a true FIFO (first-in-first-out) queue.

The second algorithm is earliest-deadline-first (EDF), which uses information about the workflow containing the task to determine the task's priority. The highest priority task is the one which originates from a workflow whose deadline is sooner than all other ready tasks' workflows' deadlines. For tasks from the same workflow their relative priority is randomly determined.

The name of the third algorithm is least-laxity-first (LLF). Here, laxity refers to the difference between the projected finish time of the task's workflow and the workflow's deadline [20, 23]. Intuitively, this algorithm expands on the previous EDF algorithm and handles better the case of a task from a workflow whose deadline is nearest but the workflow is near completion vs. another task whose workflow deadline may be further away but which has far more tasks and work left to complete which, unless the task execution is begun sooner, will risk missing its deadline. Of course, LLF requires a model to predict when a workflow will complete, which naturally relies on what scheduling decisions will be made in the future of the simulation. Although several proposals were originally formulated and tested, the most advantageous formulation determined by simulation studies and comparing results was a simple one that merely projecting the finish time using the past rate-of-execution for the workflow's completed tasks as the expected rate-of-execution for all remaining tasks. This formula is listed in Equation 3.4 using tasks, t, from workflow, w, their CPU work requirement, C_t, and the workflow birth/generation time, w_{born}. In it tasks previously executed and completed are denoted as $t^c \in w$ and tasks not yet executed or remaining as $t^r \in w$.

$$\text{projected finish time of } w = \frac{\text{remaining work}}{\text{rate of completed work}} = \frac{\sum\limits_{t^r \in w} C_t}{\frac{\sum\limits_{t^c \in w} C_t}{\text{now} - w_{born}}} \tag{3.4}$$

The fourth algorithm is an improvement upon LLF. As LLF improves upon EDF by distinguishing between a deadline soon approaching versus a tighter deadline, the proportional-least-laxity-first (PLLF) algorithm represents laxity (the difference of projected finish time of a workflow and its deadline) as a proportion of the total amount of work required among all tasks of the workflow. Use of this proportion serves to distinguish large and complex workflows from a smaller workflow with the same laxity (under the assumption that a larger workflow will have more/longer parallel chains of tasks for which parallelism can be exploited). PLLF uses the same calculation for projected workflow finish time as LLF and same calculation of laxity, but divides it by the difference of the workflow's deadline and its generation/born time. This equation of normalized tardiness is given in Equation 3.5 where for worfklow $\boldsymbol{w}$ its normalized tardiness, $\tau_{\boldsymbol{w}}$ is based on the its actual (or projected) finish time, $f_{\boldsymbol{w}}$, deadline, $d_{\boldsymbol{w}}$, and generation/birth time, $b_{\boldsymbol{w}}$.

$$\tau_{\boldsymbol{w}} = \frac{f_{\boldsymbol{w}} - d_{\boldsymbol{w}}}{d_{\boldsymbol{w}} - b_{\boldsymbol{w}}} \tag{3.5}$$

In both LLF and PLLF algorithms, the projected finish time of workflows computed by Equation 3.4 assume some work of the workflow has previously been completed (otherwise the equation would attempt to divide by zero). In the case where the first task of the workflow is being considered for scheduling Equation 3.4 cannot be used and instead the computation of laxity and projected finish time relies on a rudimentary guess at the amount of possible parallelism which may be exploited to execute the tasks of the workflow. For results in Chapter 6 the duration of tasks is assumed to be the execution time in ideal circumstances (resources operating at 100% efficiency) and tasks which may be executed in parallel are assumed to be executed with a constant amount of parallelism (depending on the type/category of workflow). Future research into more sophisticated mechanisms for computing projected finish time of workflows with no task yet completed could be conducted.

These four foundational scheduling algorithms, FCFS, EDF, LLF, and PLLF each differ in how they prioritize the next task to be executed from the ready task queue but each share the concept of a need to determine which machine to execute upon or whether to forego starting execution until a future time. A more complex scheduling algorithm may combine the selection of task and machine as a holistic approach to making scheduling decisions. In Chapter 4 one such algorithm is presented.

Chapter 4

Cost-Minimization Scheduling Algorithm

4.1 Overview

This chapter defines a scheduling algorithm first presented by this author in [11]. The Cost-Minimization Scheduling Algorithm (CMSA) is, as its name suggests, an algorithm for making scheduling decisions based on information about tasks to be scheduled and their associated "cost." The cost function is defined by the user of the CMSA. For our studies we employ a cost function based on the tardiness (amount of deadline miss) of the workflow. Whenever the algorithm is invoked to make a scheduling decision, the algorithm must weigh whether the cost savings of starting any ready task and thus speeding up the projected completion of its workflow is more advantageous than the cumulative cost incured by slowing the progress being made on all other running tasks on the same machine (if any) by committing that machine's resources to additional load of the to-be-scheduled task.

CMSA is heuristic-based, in part because it must estimate finish times of tasks (or tasks which follow it) in order to compute the proportionate miss of their deadline, but also because it measures only the impact of scheduling a ready task versus delaying the start of the ready task (increasing the cost of its workflow by delaying its projected finish time) until an estimated future time: the earliest projected finish time of any of the currently running tasks.

Let $\mathcal{W}$ denote the set of all workflows to be scheduled for execution. For each $\boldsymbol{w} \in \mathcal{W}$ there is assumed to be a cost function, $\mathcal{F}_{\boldsymbol{w}}(\tau_{\boldsymbol{w}})$, 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 $\mathcal{F}(\boldsymbol{\tau})$, is defined by summing the costs of all workflows:

$$\mathcal{F}(\boldsymbol{\tau}) = \sum_{\boldsymbol{w} \in \mathcal{W}} \mathcal{F}_{\boldsymbol{w}}(\tau_{\boldsymbol{w}}) \tag{4.1}$$

where $\boldsymbol{\tau} = [\tau_{\boldsymbol{w}}]_{\boldsymbol{w} \in \mathcal{W}}$ and $\tau_{\boldsymbol{w}}$ is the normalized tardiness of workflow $\boldsymbol{w}$ from Equation 3.5.

Because $\tau_{\boldsymbol{w}}$ is normalized, it is straightforward to compare the relative tardiness values of workflows 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 workflow functions $\mathcal{F}_{\boldsymbol{w}}(\tau_{\boldsymbol{w}})$ are non-decreasing functions. This is a reasonable assumption in practice because a sensible SLA (service-level agreement) should not allow greater tardiness to be less costly than any lesser tardiness.

4.2 The Algorithm

The function of CMSA is to decide which, if any, of the "ready" tasks present in the scheduling pool should be assigned to a machine to begin execution. Recall from Chapter 3 that scheduling decisions are allowed only when one of two events occur in the system: (1) when a task finishes execution or (2) when a new workflow arrives in the system and its first task is placed in the ready queue. During the time period between two such consecutive events, the currently executing tasks continue executing and the states of the machines, executing tasks, and tasks in the queue do not change. Regarding the state of the machines, specifically, based on the machine resource model described in Section 3.4, the efficiency value, e, of each machine does not change during the time period between consecutive events.

At each scheduling instance, and for each ready task in the queue, CMSA decides whether to start a task on a machine, based on the outcome of cost function analysis. Specifically, the scheduler estimates the cost associated with starting a ready task immediately or holding the task in the queue until the next soonest time at which any other task currently executing on the machine will finish. 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 workflow's cost, $F_{\boldsymbol{w}}(\tau_{\boldsymbol{w}})$, is estimating the finish time, $f_{\boldsymbol{w}}$, of the workflow. Recall from Eq. 3.5 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 workflows. Fig. 4.1 visually represents the amount of work remaining to be done on an individual task r (initially C_r). As is apparent from Fig. 4.1, the issue of how to best estimate the finish time,

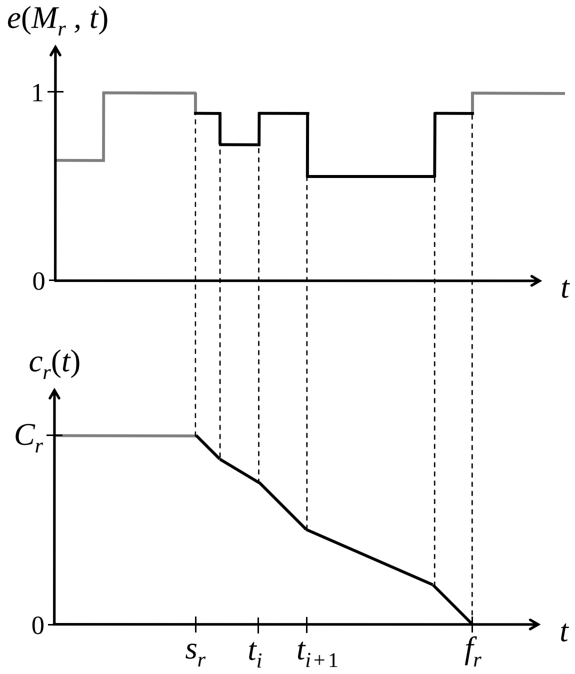

Figure 4.1: Example depiction of the effect of machine/resource efficiency on task finish time.

f_r, of even a single task is not obvious because its value 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 such as before time t_i and between t_i and t_{i+1} because of other tasks which may be started or completed on the same machine during the processing of r.

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 3.6 and Equation 3.4.

Let $\mathcal{M}$ denote the set of machines and $M(t_i)$ denote the set of tasks currently executing on machine $M \in \mathcal{M}$ at time t_i. Let $R(t_i)$ denote the set of ready tasks in the queue at time t_i, and let $\boldsymbol{w}(r)$ denote the workflow associated with task r.

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

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

The value of t_M is defined to be the next (soonest) time at which one of M's executing tasks will complete. The value of t_M is itself dependant upon whether a particular ready task r^* is started at time t_i. The formulas for the two possible values of t_M, denoted t_M^{wait} and t_M^{start}, are given by:

$$t_M^{\text{wait}} = t_i + \min_{r \in M} \left\{ \frac{c_r(t_i)}{U_r e^{\text{wait}}} \right\} \tag{4.2}$$

$$t_M^{\text{start}} = t_i + \min_{r \in M \cup \{r^*\}} \left\{ \frac{c_r(t_i)}{U_r e^{\text{start}}} \right\} \tag{4.3}$$

where $e^{\text{wait}} = e(M(t_i), t_i)$ is the efficiency of machine M if the decision is made to wait to start r^* and $e^{\text{start}} = e(M(t_i) \cup \{r^*\}, t_i)$ is the efficiency of machine M as if the decision is made to start executing r^*.

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:

$$\digamma_{r^*,M}^{\text{wait}} = \digamma_{\boldsymbol{w}(r^*)} \left(\frac{\tilde{f}_{\boldsymbol{w}(r^*)} + \Delta t^{\text{wait}} - d_{\boldsymbol{w}(r^*)}}{d_{\boldsymbol{w}(r^*)} - b_{\boldsymbol{w}(r^*)}} \right) + \sum_{r \in M} \digamma_{\boldsymbol{w}(r)} \left(\frac{\tilde{f}_{\boldsymbol{w}(r)} - d_{\boldsymbol{w}(r)}}{d_{\boldsymbol{w}(r)} - b_{\boldsymbol{w}(r)}} \right) \tag{4.4}$$

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

```
1   for scheduling instance $t_i$
2      minPenalty $\leftarrow \infty$, $r_{\min} \leftarrow \infty$, $M_{\min} \leftarrow \infty$
3      for each $r \in R(t_i)$
4         for each $M \in \mathcal{M}$
5            compute $\Delta\mathcal{F}_{r,M} = \mathcal{F}^{\text{start}}_{r,M} - \mathcal{F}^{\text{wait}}_{r,M}$
6            if $\Delta\mathcal{F}_{r,M} \leq 0$
7               compute $\mathcal{F}^{\text{penalty}}_{r,M}$
8               if $\mathcal{F}^{\text{penalty}}_{r,M} <$ minPenalty
9                  minPenalty $\leftarrow \mathcal{F}^{\text{penalty}}_{r,M}$
10                 $r_{\min} \leftarrow r$
11                 $M_{\min} \leftarrow M$
12     if minPenalty $= \infty$
13        exit
14     assign request $r_{\min}$ to machine $M_{\min}$
15     $R(t_i) \leftarrow R(t_i) - \{r_{\min}\}$
16     goto line 2
```

Figure 4.2: Pseudocode for CMSA.

$$\mathcal{F}^{\text{start}}_{r^*,M} = \sum_{r \in M \cup \{r^*\}} \mathcal{F}_{\boldsymbol{w}(r)} \left(\frac{\tilde{f}_{\boldsymbol{w}(r)} + \Delta t^{\text{start}} \left(\frac{1}{e^{\text{start}}} - \frac{1}{e^{\text{wait}}} \right) - d_{\boldsymbol{w}(r)}}{d_{\boldsymbol{w}(r)} - b_{\boldsymbol{w}(r)}} \right) \tag{4.5}$$

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

$$\mathcal{F}^{\text{penalty}}_{r,M} = \Delta\mathcal{F}_{r,M} + \mathcal{F}_{\boldsymbol{w}(r)} \left(\frac{\tilde{f}_{\boldsymbol{w}(r)} + \Delta t^{\text{wait}} - d_{\boldsymbol{w}(r)}}{d_{\boldsymbol{w}(r)} - b_{\boldsymbol{w}(r)}} \right) \tag{4.6}$$

Fig. 4.2 provides the precise description of CMSA. For a given scheduling instance t_i, CMSA first performs computations for all combinations of ready tasks and machines, refer to lines 3 through 11. After completing this phase of computation, CMSA then determines whether there exists a task that can be started on a machine. If the answer is no, then the algorithm exits, refer to lines 12 and 14. However, if the answer is yes, then the selected task is assigned to the selected machine (line 14), the selected task is removed from the set of ready tasks (line 15), and the algorithm again performs computations for all combinations of remaining ready tasks and machines (line 16). The complexity associated with performing computations for all combinations of ready tasks 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 tasks, thus ready tasks will tend to accumulate in the queue. 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 tasks, thus ready tasks will tend to be removed quickly from the queue. 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.

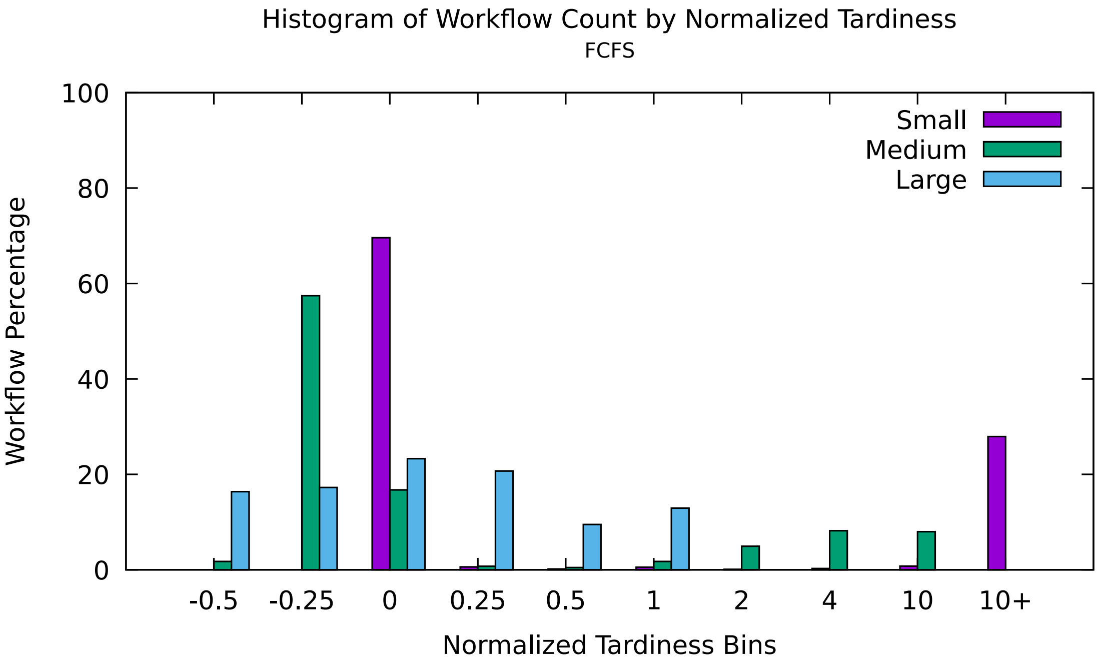

Figure 4.3: Histogram of percent of workflows completed (by workflow type) for various ranges of normalized tardiness for FCFS.

4.3 Comparison of Behavior

In order to compare the behavior of CMSA to algorithms such as FCFS and PLLF a histogram of workflows completed within various ranges of normalized tardiness by workflow type is shown for FCFS, PLLF, CMSA with a sigmoid cost function, and CMSA with a quadratic cost function in Figures 4.3, 4.4, 4.5, and 4.6, respectively.

As illustrated CMSA, with either cost function, generally performs better than FCFS or PLLF (which itself performs better than FCFS) in terms of more workflows completed at or below a normalized tardiness of zero (i.e. finished at or before their deadline). Furthermore, CMSA with the quadratic cost function keeps the maximum normalized tardiness of any workflow below that of PLLF

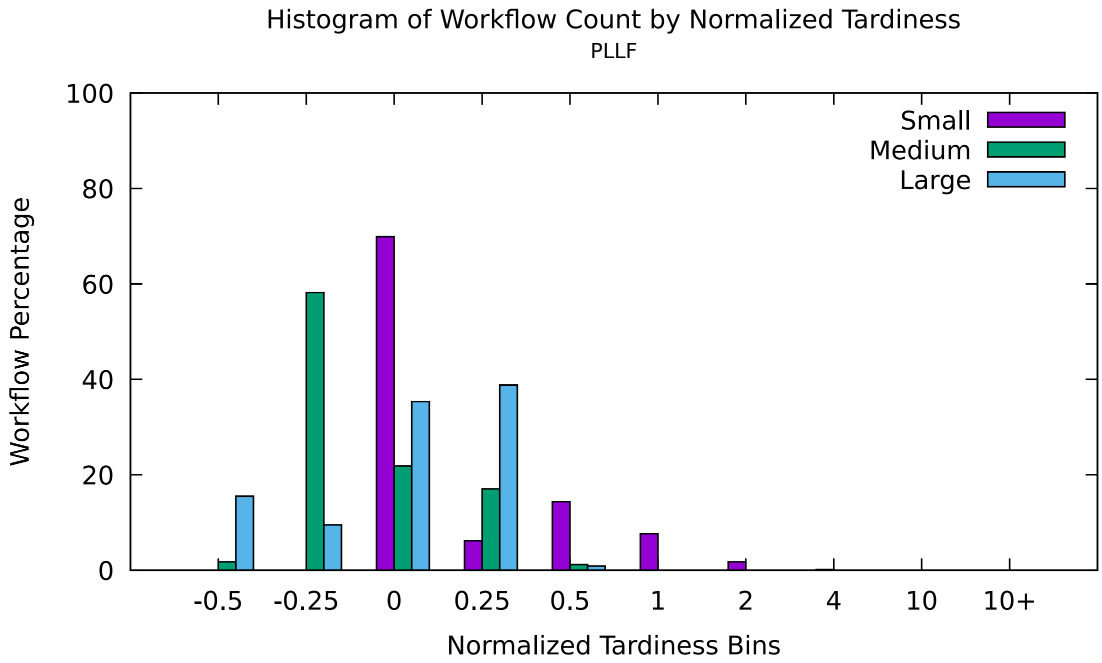

Figure 4.4: Histogram of percent of workflows completed (by workflow type) for various ranges of normalized tardiness for PLLF.

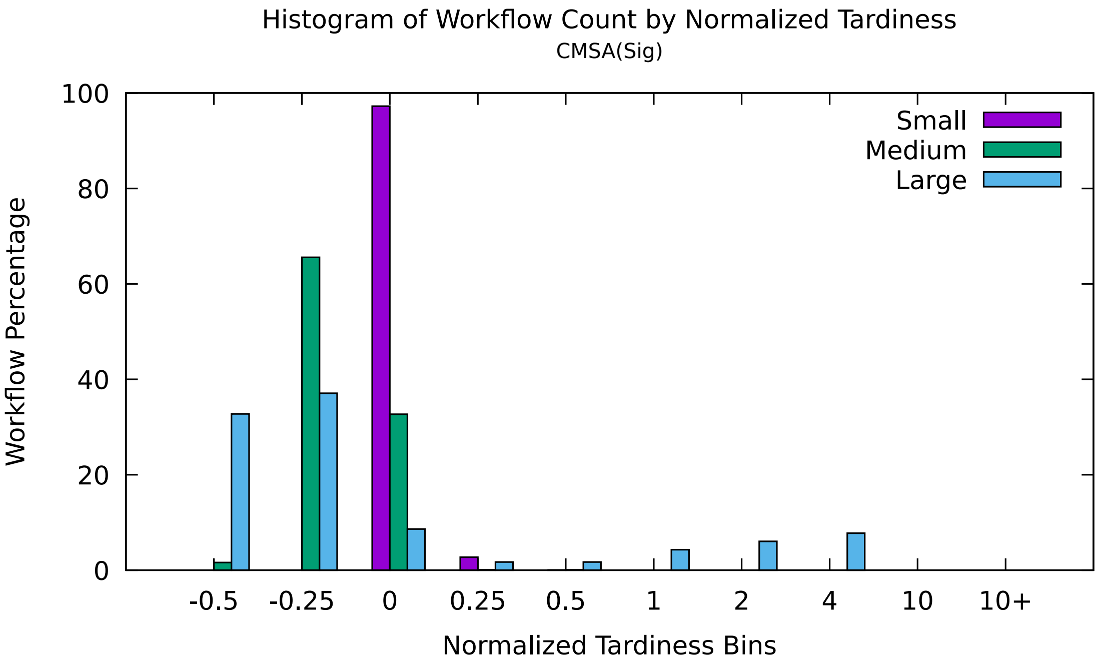

Figure 4.5: Histogram of percent of workflows completed (by workflow type) for various ranges of normalized tardiness for CMSA with a sigmoid cost function.

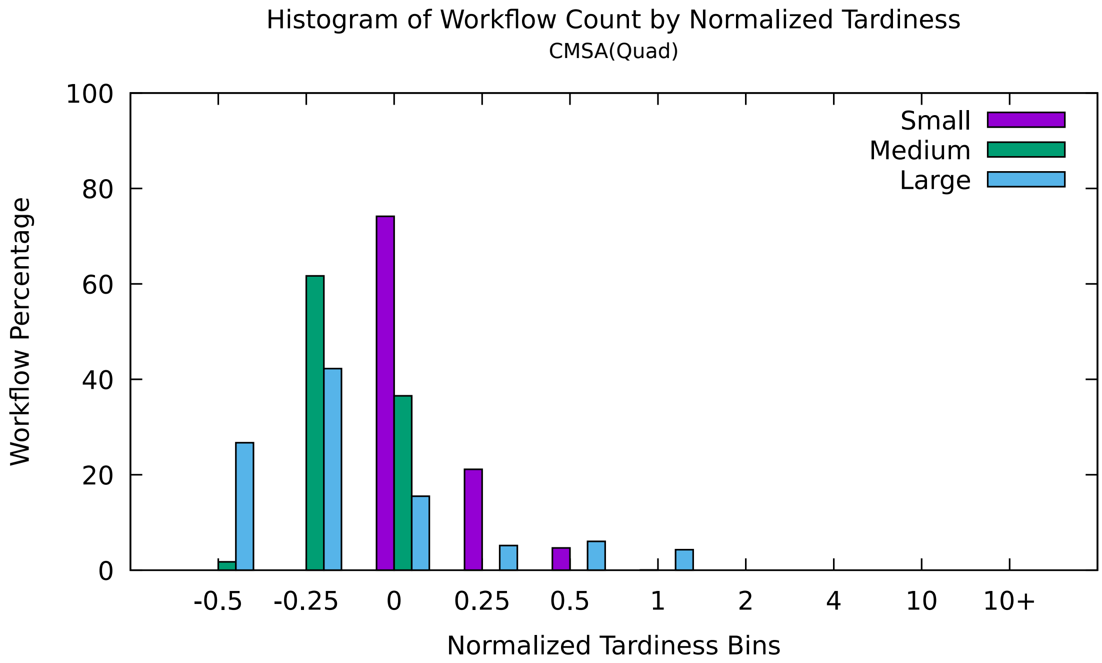

Figure 4.6: Histogram of percent of workflows completed (by workflow type) for various ranges of normalized tardiness for CMSA with a quadratic cost function.

(and FCFS). However, the results also show distinct differences in how CMSA performs generally for each workflow type.

First, where FCFS and PLLF tend to cause small workflows to be completed the latest (highest normalized tardiness), CMSA for either cost function tends to perform much better for small workflow and instead let some large workflows finish latest. Furthermore, CMSA tends to finish the majority of small workflows just at or before their deadline (i.e. a normalized tardiness of 0 or below).

Second, CMSA for both cost functions achieved roughly the same outcome for medium workflows (the majority completed 25% or more before their deadline and the rest completed between 0% and 25% before the deadline). PLLF achieves a similar result though with about 20% of medium workflows up to 50% after the deadline. FCFS also completes a majority (nearly 60%) of medium workflows early but with about 20% of medium workflows several times later than their deadline.

Finally, one of the biggest distinctions in behavior is the treatment of large workflows. FCFS tends to complete nearly equal amounts of large workflows at all intervals between 50% early and 100% late. PLLF keeps large workflows below 50% late with the majority completed between 25% early and 25% late. CMSA's treatment of large workflow differs depending on the cost function. The sigmoid cost function, because it assigns nearly equal cost to a late workflow vs. a very late workflow, tends to cause CMSA to essentially "give up" working on a workflow expected to be completed "too late" while other workflows are present in the system (which can benefit from being prioritized and possible go from being completed late to being completed early). As a result, CMSA with the sigmoid cost function tends to complete most large workflows early or on time, but a few large workflows are "neglected" until the system is loaded lighter and thus are completed up to four times later than their deadline. CMSA with the quadratic cost function, on the other hand, treats workflows as more costly the later their completion becomes and thus tries to minimize the worst tardiness of all workflows. As a result only a small number of large workflows are completed up to 100% late.

Chapter 5

Framework for Evaluating Scheduling Robustness

5.1 Overview

Robustness of a scheduling algorithm to the presence of error can be measured in two ways. The first is to quantitatively measure the difference in scheduling decisions the algorithm makes with and without error. This may be measured as the difference in order the algorithm chooses to begin execution of tasks (i.e., because of error the algorithm begins execution of task t_1 ahead of task t_2 where if no error were present the opposite order would be selected). Difference in scheduling decisions can also be measured by examining the actual times when tasks begin executing. By this measure of robustness, however, a simple algorithm such as first-come-first-served (FCFS) may be considered perfectly (or maximally) robust because the presence of error in the task requirements does not change the order in which the workflows arrive and thus their starting and subsequent tasks are add to the queue (recall from Section 3.6 that FCFS treats the queue of ready tasks as a FIFO data structure and scheduled tasks in the order they were placed in the queue). On the other hand, for other scheduling algorithms, even though different scheduling decisions may be made due to model error the performance objective of scheduling algorithms (e.g., to complete workflows by or ahead of their deadline) may still be achieved (i.e., many potential static schedules may achieve the same objective).

Measuring the difference in performance objective in the presense of model error vs. an error-free model is the second means for measuring robustness. Because this research is focused on dynamic scheduling algorithms and not static schedules and also focused on practical outcomes, this second approach of defining robustness is the one adopted in this research.

Multiple potential performance objectives are practically considered in other research such that scheduling algorithms' robustness with respect to each may be measured and compared with and without the presence of error in the model. Such objectives include:

- Percent of workflow completed late (after their deadline)
- Average and Maximum tardiness of workflows
- Average cost (e.g., as defined in Chapter 4)
- Latest completion time of all workflows (i.e., makespan)

In the present research, results will focus primarily on the percent of workflows completed early vs. late (and also at various proportions of earliness and lateness) as well as maximum normalized tardiness as the objectives against which robustness to error will be measured.

5.2 Actual Platform Feedback

As results in Chapter 6 will show one of the most important aspects in achieving robustness is the utilization of feedback from the "actual" platform to the "model" platform. Consider a simple example scenario where two tasks, t_1 and t_2, are to scheduled for execution in a platform with only a single machine. The resource requirements of each task is substantial enough that to execute both tasks concurrently would result in an efficiency of less than 50% the efficiency of executing either task alone. It would thus generally be best to execute the tasks sequentially to achieve any reasonable objective (e.g., shortest makespan, lowest percent of tardiness). Because of the presense of error, however, the "model" platform will model the first scheduled task as completing at a different time than it does in the actual platform. In the case that the modeled task completion time is later than the actual platform, feedback of the actual task's completion prevents the actual machine from wastefully being idle because in the "model" platform the task is still executing. Figure 5.1 illustrates a scenario such as this one and the effect on "model" and "actual" machine efficiencies as well as how the remaining work (C_1 and C_2) of both the "model" and "actual" tasks, t_1 and t_2 are decreased over time. This kind of scenario will lead to generally worse performance for scheduling algorithms as they make poor use of resources by underutilizing them (e.g., the period in Figure 5.1 where "actual" machine efficiency is at 1.0.

In the other case the modeled task's completion time is earlier than the actual platform. This scenario is generally worse because without feedback from the actual platform the model platform's newly idle machine will lead the scheduling algorithm to decide to begin execution of the second task. This further exacer-

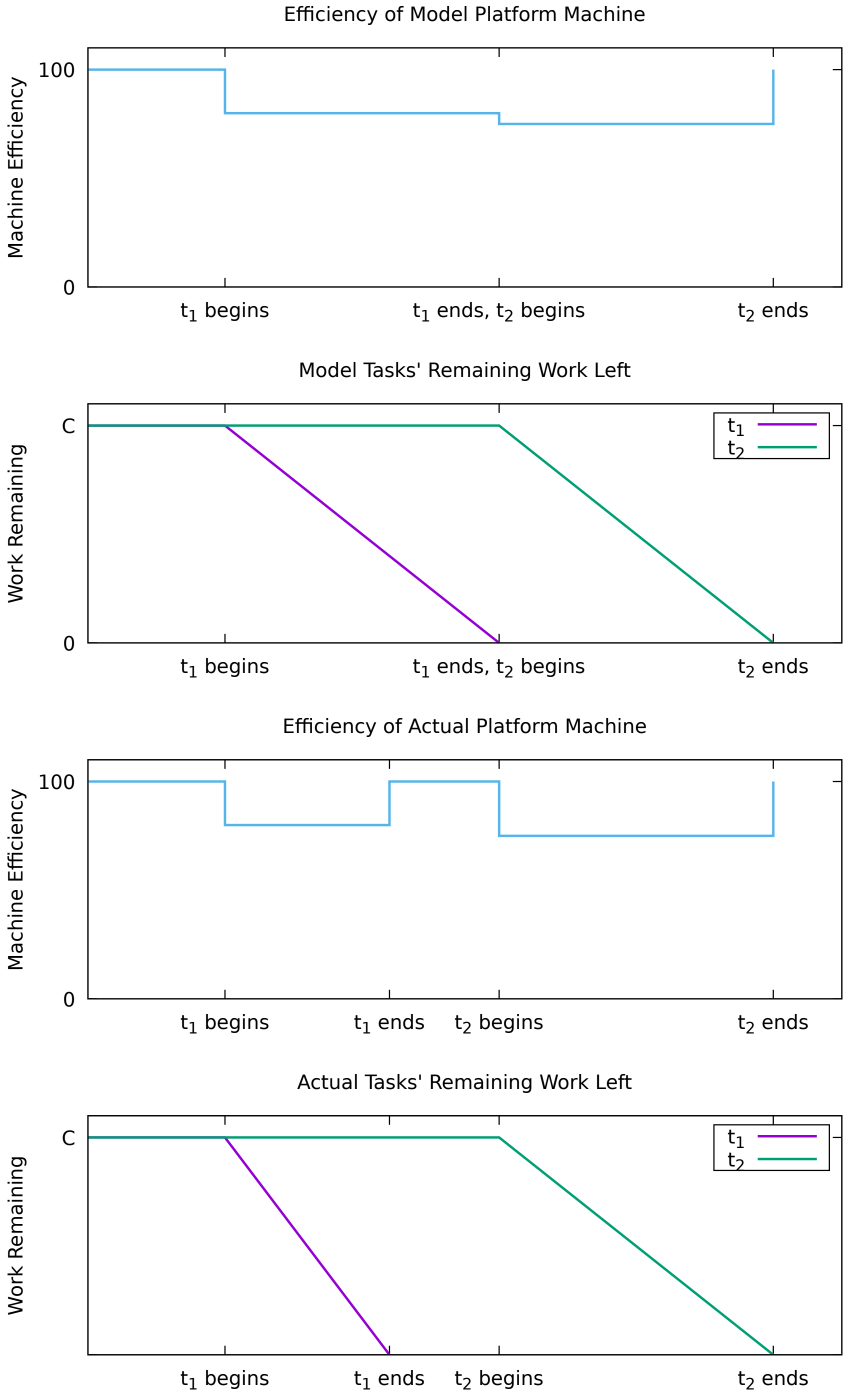

Figure 5.1: Effect on "actual" machine efficiency and remaining task work when "model" error over-estimates task requirements and thus models the task as completing later than it does in the "actual" platform.

bates the problem by deviating the modeled completion time and actual completion time of the first task because the concurrent execution of both tasks (for any amount of time) leads to lower efficiency of the machine and the actual first task requiring longer to complete execution. The same lower efficiency causes the same exaggerated difference between the second task's modeled and actual completion time. An example of this scenario is illustrated in Figure 5.2. Furthermore, in more elaborate scenarios with more tasks this second kind of problem becomes a sort of negative feedback loop creating more and more deviation between the model and actual platforms in terms of which tasks are executing vs. completed and the tasks' completion times.

The primary solution to this problem is to create a system whereby the completion of tasks on actual machines can be fed back to inform the model platform of tasks' (actual) completion. If this feedback is guaranteed for all tasks, the notion of tasks completing in the model platform can be removed or ignored. However, if anything less than complete feedback occurs, even though it may be 99+% of tasks' completions from the actual platform, then the model platform's completion of tasks must be used to approximate the actual task completing. Otherwise, a model platform task be modeled as executing indefinitely in the case that the completion time of the corresponding actual task is not fed back to the model.

This distinction between a model and actual platform, where decisions made by a scheduling algorithm are manifested (by a component known as a Task Assigner), is illustrated in Figure 5.3. The potential presence of feedback from the actual platform to the model platform is represented by a dashed line.

A secondary approach to dealing with the effects of error would be to avoid or decrease the likelihood of the scenario in which the model platform models

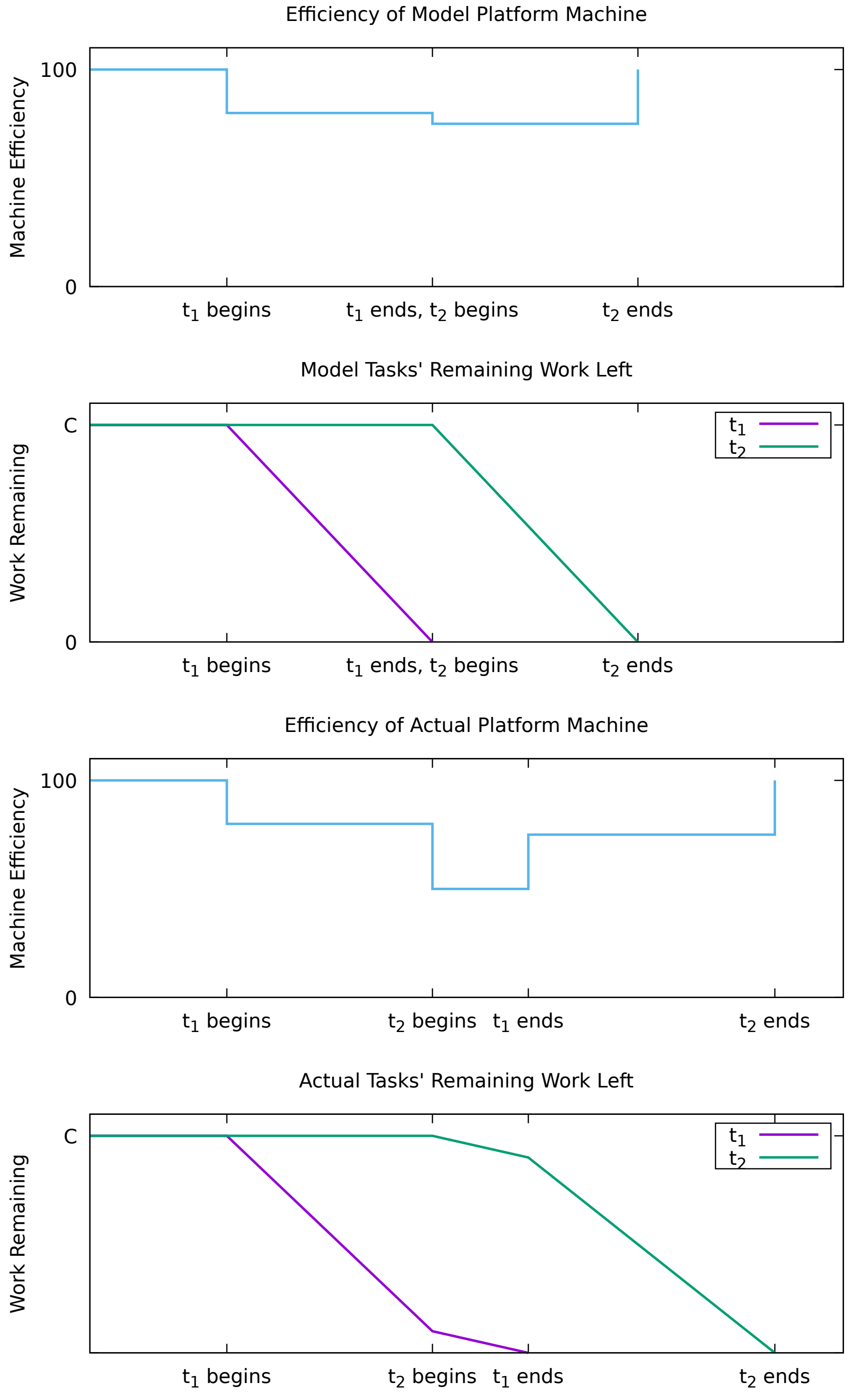

Figure 5.2: Effect on "actual" machine efficiency and remaining task work when "model" error under-estimates task requirements and thus models the task as completing earlier than it does in the "actual" platform.

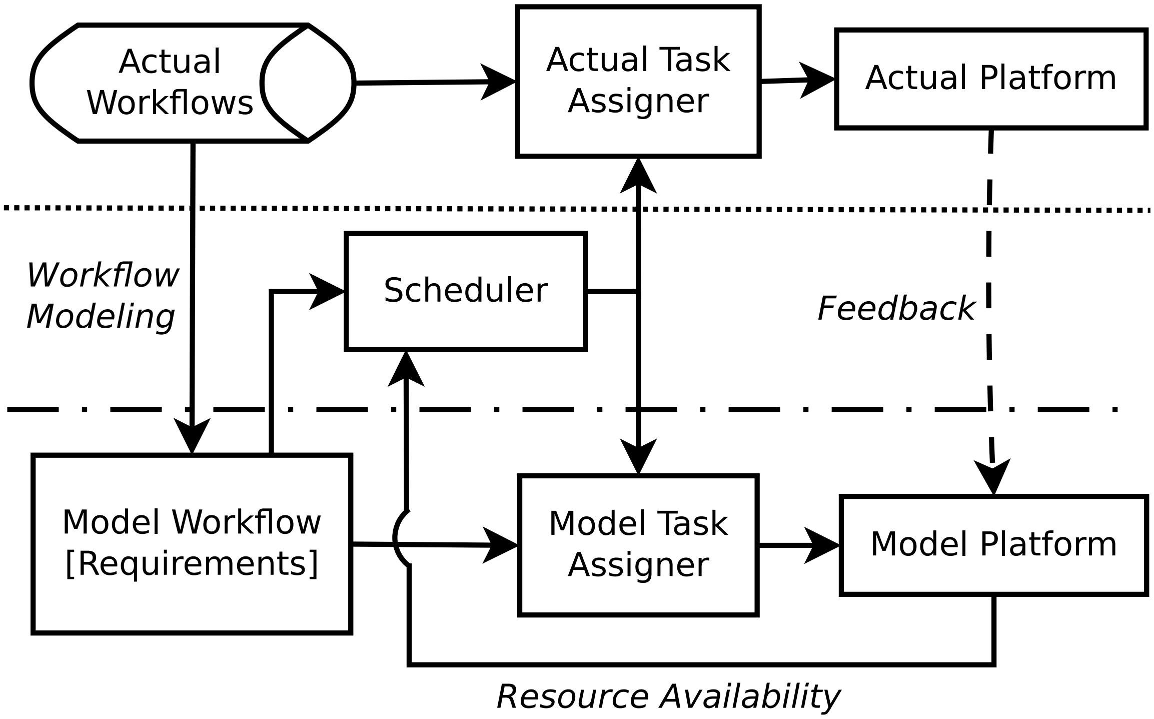

Figure 5.3: Block diagram from [10] illustrating components of proposed framework for evaluating effect of model error and scheduling algorithms robustness.

the task completion as being earlier than the corresponding task in the actual platform. In effect, this means that the error in the model would favor overestimating task resource utilization, for example, the amount of work required to complete execution of each task. The following section outlines the approach in this research for dealing with error that may underestimated task requirements.

5.3 Applying Error Bias

It is unrealistic to know the magnitude of error inherant in the "model" platform because such knowledge would imply the error could simply be negated to achieve a perfectly accurate model. In this research little is assumed about the nature (overestimate or underestimate) or magnitude of the model error and a generic approach is applied in order to bias the model's error such that it is less likely or even guaranteed not to underestimate tasks' resource utilization or work requirement.

Let $\hat{X}$ denote a modeled or estimated value of an actual value, X, but with some "error" making it generally inaccurate. There are several biasing approaches in which an alternative estimate of X can be computed from $\hat{X}$ such that this alternative estimate, $\hat{X^b}$, is less likely than $\hat{X}$ to underestimate X. The simplest such approach would be to add a constant value, C, to $\hat{X}$, as in Equation 5.1. In order to choose an effective value for C, however, information about the magnitude of X itself as well as the maximum amount of error in $\hat{X}$ is needed. In Figure 5.4 the effect of various C values is demonstrated where X is a term having error, bound parameter e, applied to it from a triangle-shaped distribution between $[-e, e]$. To eliminate the potential of underestimating the true value while minimizing the amount of overestimation the value $C_1 = e$ is illustrated

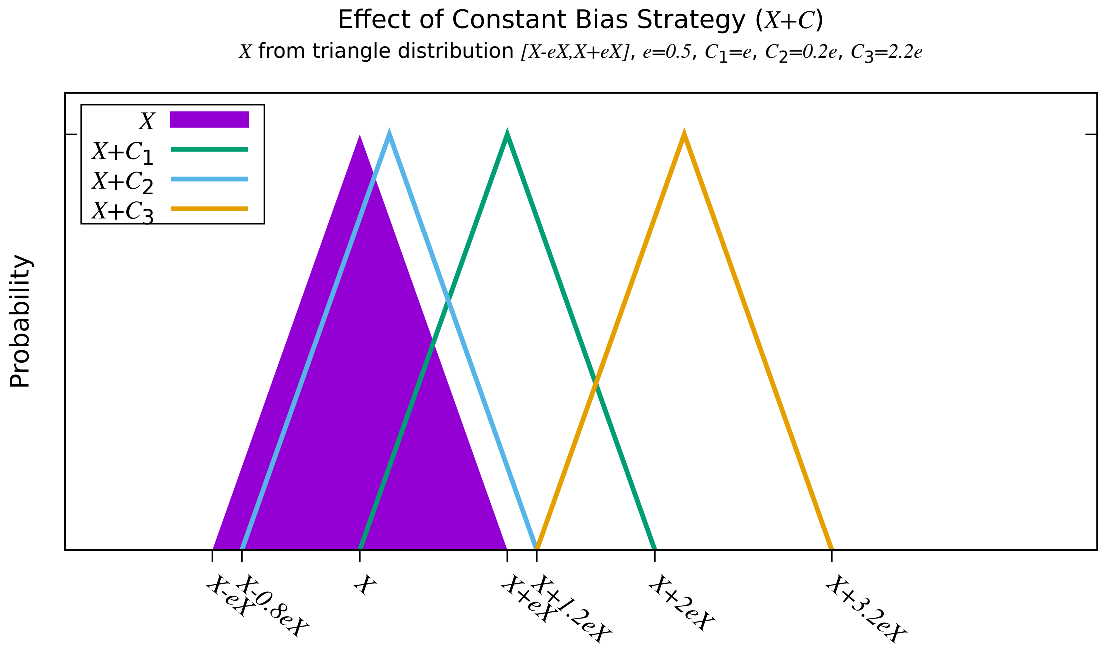

Figure 5.4: Illustration of probabilities for value X with error bound term, e, from triangle distribution applied, along with several C constant bias terms applied.

as the green line. A value of C too small such as the case of $C_2 = 0.2e$ results in probability that the biased term still underestimates the true value, X (refer to the blue line). A value of C too large such as the case of $C_3 = 2.2e$ (refer to the orange line) may overestimate more than necessary and even result in the biased term having bounds that don't intersect with the original value, X. Thus, using a constant value for biasing the model value is generally only effective if the magnitude of the error is known or estimated with near accuracy.

$$\hat{X}^b \overset{1}{\hookleftarrow} \hat{X} + C \tag{5.1}$$

If the bounds of the value of the error are not known but the bounds of the proportion (or percentage) of error is known or reasonably estimatable as $\hat{e}$, then

$\hat{X}$ can be normalized as in Equation 5.2. This bias strategy will in the rest of this research be referred to as the proportionate bias strategy.

$$\hat{X^b} \xleftarrow{2} \hat{X}/(1-\hat{e}) \tag{5.2}$$

For example, if $\hat{X}$ is known or assumed to be within 10% (i.e., $e = \hat{e} = 10\%$) of the true value, X, then $\hat{X} \in [0.9X, 1.1X]$. By dividing $\hat{X}$ by $1 - \hat{e}$ as in Equation 5.2 the normalized value $\hat{X^b} \in [X, 1.222X]$. Given an accurate estimate of the error's bounds as a percentage of X, the distribution of $\hat{X^b}$ can be shifted to never underestimate X. Unlike the constant bias strategy, however, the distribution bounds of $\hat{X^b}$ are also "stretched" in addition to being shifted (i.e. why the $\hat{X^b}$ distribution above may overestimate X by not just e as $\hat{X}$ did, but by $\frac{1+e}{1-\hat{e}}$). In other words, while the width of the distribution of $\hat{X}$ is $2e$, the effect of the proportionate bias strategy produces a value, $\hat{X^b}$, whose distribution width is $\frac{1-e}{1-\hat{e}} + \frac{1+e}{1-\hat{e}}$. This effect is illustrated in Figure 5.5 where estimates (d) for the error term, e, as a percentage of X are shown. Again a triangle distribution is used for the probability distribution of values $\hat{X} \in [X(1-e), X(1+e)]$ as an example.

The main benefit of the second bias strategy over the first is that the error estimate may be given as a percentage of the true value, X, as opposed to a constant value. Additionally, an accurate estimate perfectly eliminates the propability of having an underestimated value $\hat{X}$. The disadvantage of the second bias strategy is the stretching effect on the distribution bounds, which worsens with larger error terms. As a compromise, another bias strategy is to use an estimate, $\hat{e}$, of the error term e again expressed as a percentage of X as in Equation 5.3:

$$\hat{X^b} \xleftarrow{3} \hat{X}(1+\hat{e}) \tag{5.3}$$

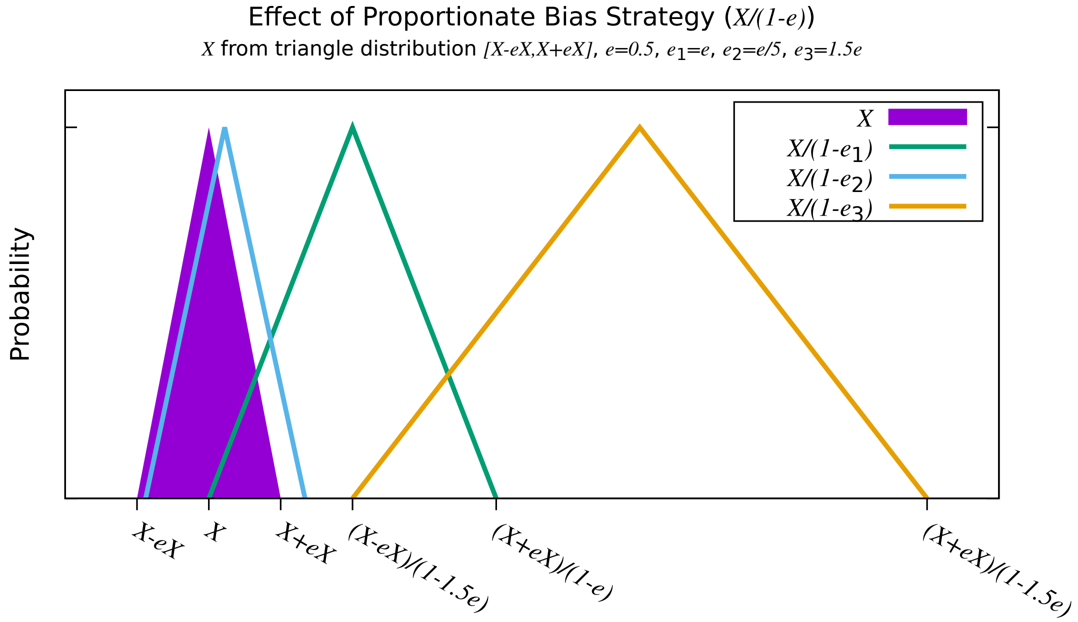

Figure 5.5: Illustration of probabilities for value x with error term, $e = 0.5X$, from triangle distribution applied, along with several estimates of e as a percentage of X applied using Eq. 5.2: $e_1 = e = 0.5$, $e_2 = 0.1$, $e_3 = 0.75$.

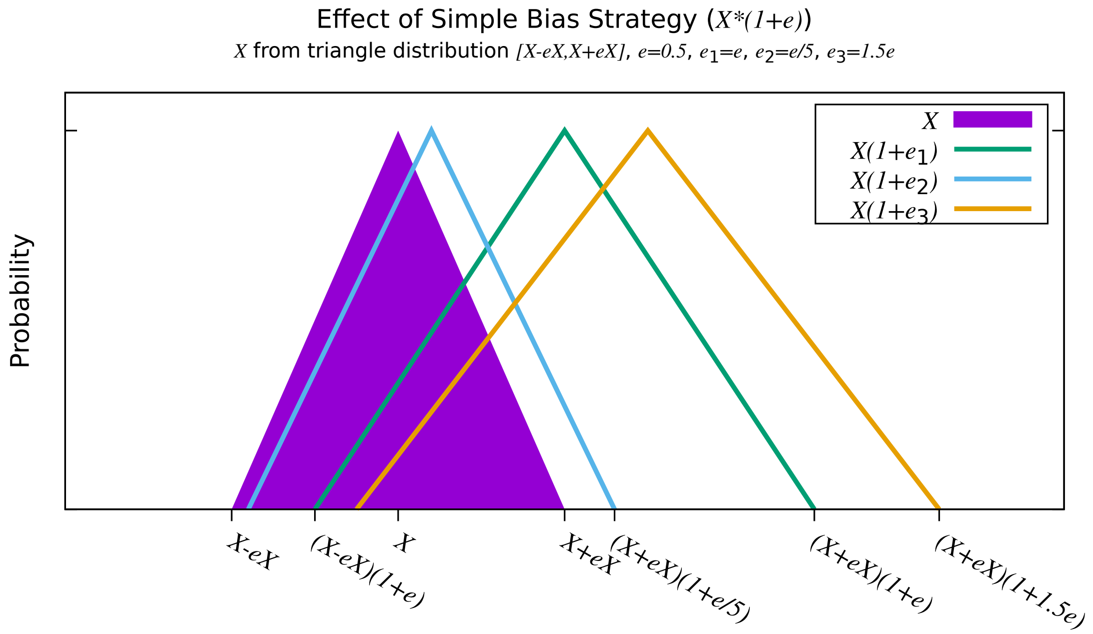

Figure 5.6: Illustration of probabilities for value X with error term, $e = 0.5$, from triangle distribution applied, along with several estimates for e as a percentage of X applied using Eq. 5.3: $e_1 = 0.5$, $e_2 = 0.1$, $e_3 = 0.75$.

This simpler bias strategy also works reasonably well for smaller (0-10%) error terms with less stretching than the second bias strategy. It does still underestimate the true value X even with a perfect estimate, $\hat{e}$, of e, but by a relatively small amount. In fact, for the simple bias strategy to produce an $\hat{X}^b$ which does not underestimate X requires an $\hat{e} = \frac{e}{X-e}$, which is impractical to know since it requires knowledge of X. The expanded distribution of $\hat{X}$ with the simple bias strategy is $(1 - e)(1 + \hat{e}) + (1 + e)(1 + \hat{e})$. Figure 5.6 demonstrates the effect of using this third bias strategy under the same conditions of a triangle distribution where $e = 0.5$ and the three error estimates, $\hat{e}$ are 0.5, 0.1, and 0.75 as in Figure 5.5.

In Chapter 6 the presence of feedback (none, full, and various levels of partial) and each of these three bias strategies (as well as no bias) are simulated and their effects on each scheduling strategy from Section 3.6 in the presence of various levels of model error bound values are presented.

Chapter 6

Numerical Studies

6.1 Overview

All results that follow are based on a software simulator developed for this and related research. This simulator represents a distributed system and the arrival of various types of workflows as modeled and described in Chapter 3. The quantities and associated parameterization of the resources and machines are configurable, as are the size and “shape” of workflows, their tasks’ resource requirements, and arrival rates. This simulation of a distributed system is modeled after a real world system developed and in use at the author’s place of full-time employment as a distributed system framework architect and software engineer. The parameterized workflows, arrival rates, etc. are modeled after a typical 24-hour period of processing for this real world system.

The machines simulated were a reasonably small distributed system topology of sixteen machines each with identical resource capacities. The workflows were modeled as three “classes” of workflow representing essentially small, medium, and large jobs. These sizing designations apply to both the overall size of the workflow instances in terms of number of tasks, as well as the resource utilizations of the tasks. Additionally, the arrival rates were distinct among each of the three workflow types and these rates themselves for each workflow type varied over a simulated 24-hour period to mimic typical loadings of the real system. The small-sized workflow are representative of jobs from an interactive application in which job size and requirements are constrained to be small by design in order to provide a responsive experience to users. These workflows arrive according to the overlap of three Poisson distributions representative of the working hours of users in three different geographic regions of the world.

The medium-sized workflows were roughly a factor of three to five times larger than the small-sized workflows. Additionally, the tasks' resource (CPU and Memory) utilizations were similar to other workflow types but the amount of work/time required to complete the tasks were an order of magnitude larger than the small-sized workflows' tasks. The arrival rate of these workflows was a simple Poisson distribution over the entire 24-hour simulation period. The large-sized workflows were again another order of magnitude larger in terms of task work/time required to complete, but similar in terms of the resource utilizations. The size of large-sized workflows were a factor of four to ten times larger than the medium-sized workflows. Their arrival rate was based on a exponential decay rate representative of large jobs that generally begin arriving shortly after availability of new daily data or functionality (which in this simulated scenario occurs at time $25,200$, roughly 30% into the simulation's 24-hour, or $86,400$ time units, period) with some jobs sporadically arriving later throughout the simulation.

The simulated deadlines for workflows was computed using a random factor of 10-30% beyond a computed expected runtime of the workflow based on a reasonable assumption of concurrent execution of tasks (2-5 for workflows sized large enough to have concurrency which could be exploited) and on a lightly loaded collection of machines.

For all simulations, the same set of workflows generated according to the descriptions above were used, comprised of 4,354 workflows: 2,280 small-sized, 1,958 medium-sized, and 116 large-sized workflows. The effect of the arrival rates and summary of the arrivals of each of the 4,354 workflows is depicted in Figure 6.1. As evidenced by the graph this simulated scenario provides multiple time periods of varying load demands on the distributed system including times near the beginning and end when only medium-sized workflows are arriving in the system (though near the end there is likely still some previously-arrived large workflows still being processed), a time period when both medium- and small-sized workflows are arriving, and a time period when all three sizes of workflows are arriving including times when small-sized workflows are arriving at double their normal rate (times when normal business hours of two geographic world regions overlap).

To study the effect of "error" in the model of workflows and tasks used by the scheduling algorithms relative to the actual values simulated, different levels of error were applied to some of the resource requirements for all tasks. For one set of simulations only the resource requirement of the amount of CPU cycles the task must complete executing has an "error" term applied. This resource requirement represents the amount of work each task has to execute, which for many applications is based on the inputs inherent in the task for the various algorithms and computations it will perform. Such data dependence can be difficult to either measure or predict. In a second set of simulations all the resource requirements of all tasks had "error" terms applied to them: the CPU cycles, the CPU load factor, and memory footprint of the task. Where results are

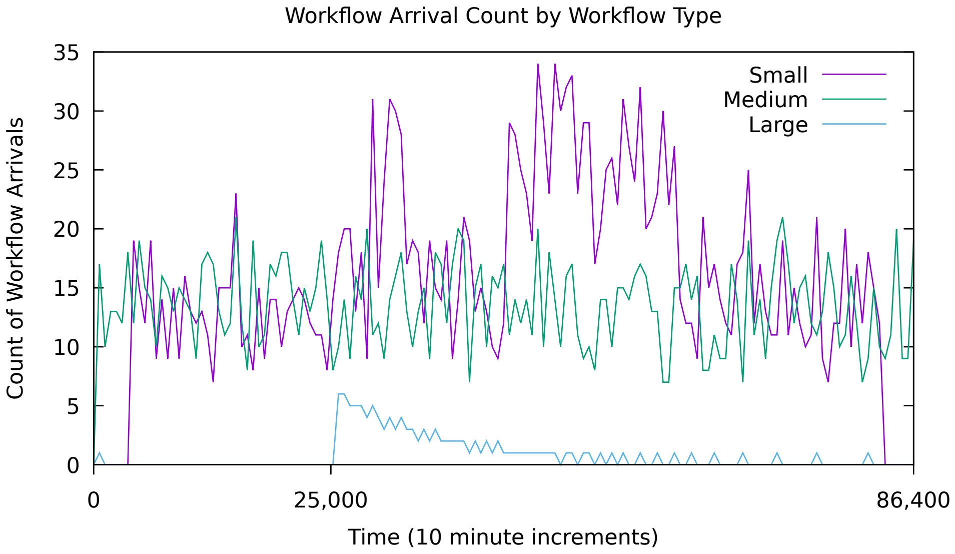

Figure 6.1: Histogram of workflow arrivals by "class."

presented, the types of resource requirements with error applied and the nature of that error's stochastic distribution are stated.

The application of error was to apply a value taken from a random distribution to each task's resource requirement(s) as a percentage of the original value. For example, to simulate a small amount of error in each tasks' modeled CPU work cycle requirement the modeled requirement value would be equal to the actual requirement value plus or minus a percent taken from a random distribution between -1% and +1% (simulations were conducted for a uniform distribution). Numeric studies performed varied this error bound from 0.1% up to 50%. In this research all errors terms were centered on 0.0, meaning that the error models used are unbiased. Because the source and cause of error may generally be unknown, there is no reason to assume it either underestimates or overestimates the true value more often than the other.

Error Bound	**% Feedback**	**Bias Type**
0.1%	100%	None
0.5%	99.9%	Constant Bias
1%	99.5%	Simple Bias
5%	99%	Proportionate Bias
10%	95%	
50%	90%	
	50%	
	0%	

Table 6.1: Table of simulation characteristics evaluated. For each cross-product of values ten simulations with different seeded error terms was executed.

6.2 Dimensions of Perturbation

In order to simulate and study the effect of multiple aspects or characteristics of the model subsystem, a cross-product of simulations are executed. For each intersection of values for all aspects ten separate simulations are executed each with a different seed value to affect the random number distribution used as the error terms applied to each tasks' resource requirements. Results presented then are an average across these ten simulations, in order to reduce the likelihood that any single set of random values skewed simulation outcomes in a way that may effectively skew the conclusions drawn.

As discussed earlier in this chapter one of dimensions of study is that of the potential magnitude of the amount of error. Another aspect varied was that of the type of random distribution from which error terms were drawn. The concept of feedback from "actual" to "model" platform discussed in Section 5.2 was another aspect varied amount different percentages of actual task completion events which were used to correct the model. A summary of these dimensions and values for each one for which simulations were executed is summarized in Table 6.2.

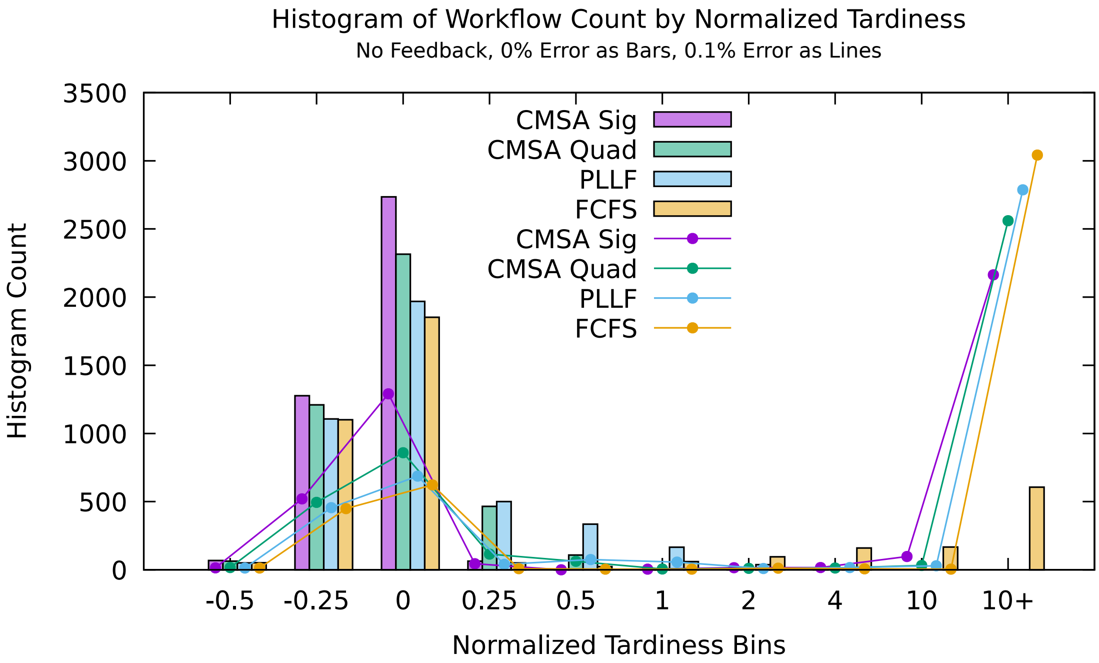

Figure 6.2: Effect on the histogram of workflows completed by normalized tardiness without "actual" system feedback of task completions in the presence of small error (taken from a uniform distribution and applied only to the CPU work requirement of tasks) for all scheduling algorithms.

6.3 Results Concerning the Impact of Feedback

As first presented in [10] and discussed in detail in Section 5.2, the most negatively impactful aspect of model error for all scheduling algorithms is a lack of feedback from the "actual" system about the completion of tasks. As depicted in Figure 6.2, even the presence of a small amount of error (0.1%) in only the CPU work requirement of tasks causes all four scheduling algorithms to finish significantly fewer workflows on time. Specifically, the lack of feedback results in a significant number of workflows completing 10-100 times later than their deadline (comparing the line graph to the bar graph of the same color).

As discussed in Section 5.2 this effect of the lack of feedback, because some

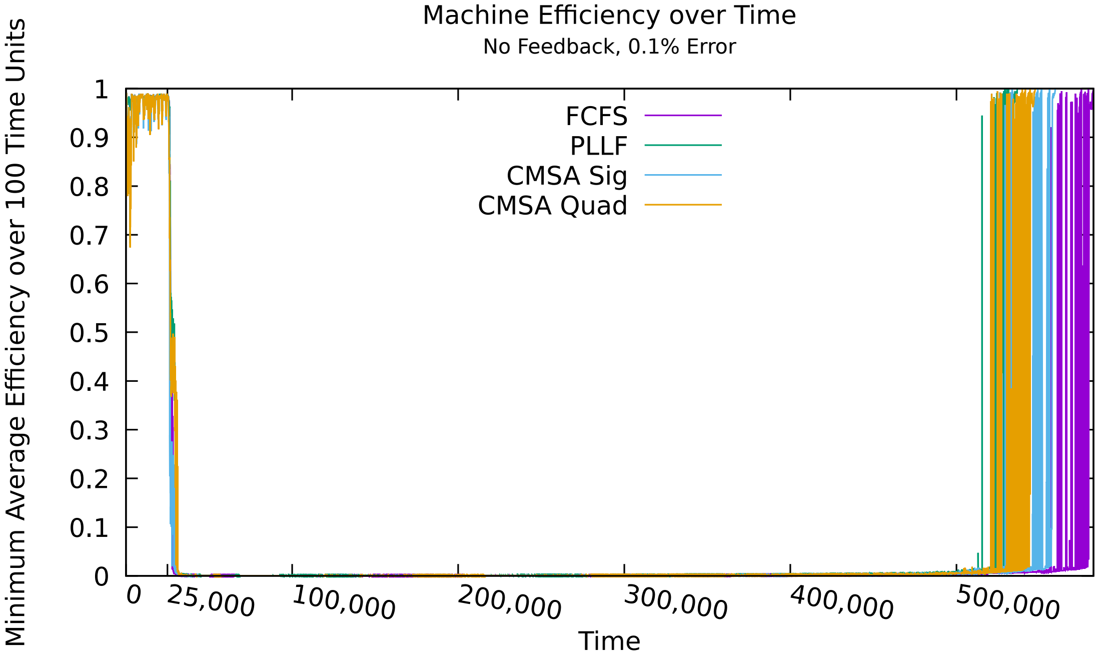

Figure 6.3: Minimum machine efficiency over time for all scheduling algorithms in the presence of small (0.1%) error in the CPU work requirement of tasks and no feedback.

tasks are modeled as completing before they truly have, causes all algorithms to overload machines because they are modeled as having fewer executing tasks, but by beginning execution of more tasks, loading machines heavier, and decreasing their efficiency, the model deviates even further from the “actual” system. This is illustrated in Figure 6.3 where the minimum machine efficiency once the system begins being loaded (once large workflows begin arriving around time 25,000) drops to nearly 0% until long after workflows have stopped arriving in the system. Contrast this with Figure 6.4 that graphs minimum machine efficiency with no modeling error present (in which case feedback is irrelevant).

From Figure 6.4 notice how both FCFS and PLLF scheduling algorithms have a limit of minimum machine efficiency at 70%. This is because both algorithms

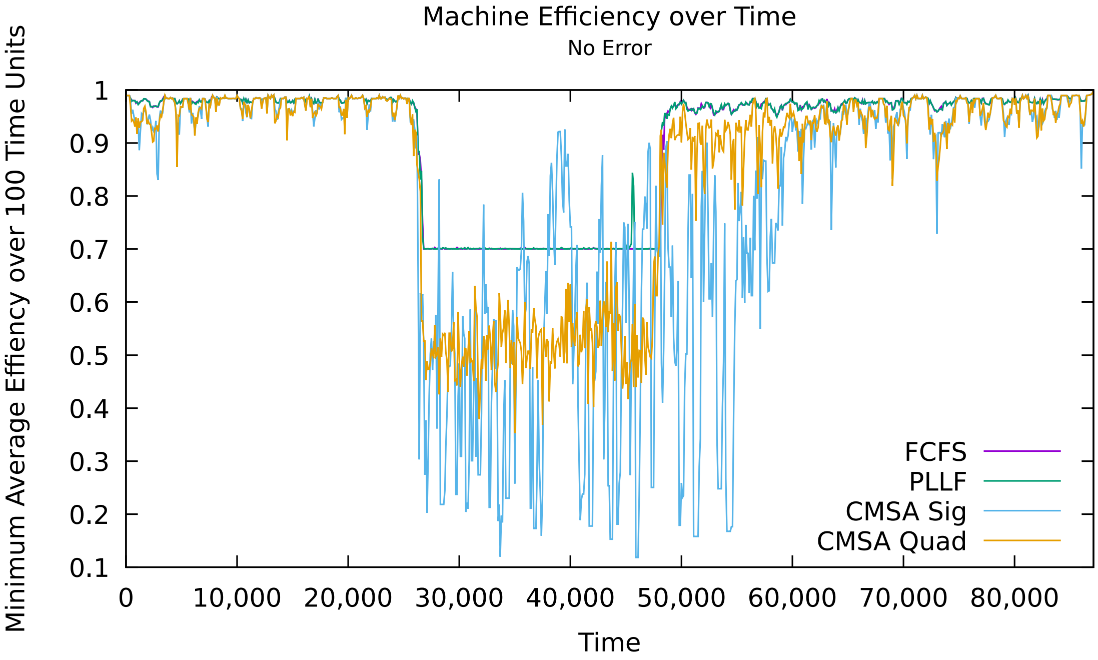

Figure 6.4: Minimum machine efficiency over time for all scheduling algorithms when no modeling error is present.

only prioritize which ready task to begin executing next, and must be paired with a policy of when to stop scheduling tasks to begin executing based on a pre-defined machine efficiency threshold. Thus an arbitrary limit to how low machine efficiency can become before no more ready tasks will be assigned to that machine is chosen, which from [24] was determined to be optimal at around 70%. CMSA, in part, achieves better performance (with respect to percent of workflows completed late and maximum normalized tardiness, as well as other measures) by pushing machine load higher (and resource efficiency lower) for at least some periods of time.

In addition because the errors in model task CPU work requirements cause all four scheduling algorithms to overload machine resources when feedback from the "actual" system isn't present to correct the model with respect to what tasks are still executing vs. completed, the time at which each scheduling algorithm finally complete execution of all tasks and workflows is often many times later than otherwise.

When complete feedback of all task completion times is utilized (i.e., the modeling of a task completion is ignored and tasks are only declared completed when feedback from the "actual" system indicates they are completed), performance of all four scheduling algorithms returns to roughtly the same level as in the case of no modeling error, as shown in Figure 6.5. Although some algorithms fail to achieve the same performance as in the presence of no (0%) error, most of the discrepancy occurs between normalized tardinesses of -0.25 up to 1.0 and is far less drastic than in the case when no feedback is employed.

This relative robustness of all scheduling algorithm to high error given feedback from the "actual" system of task completions is critical because it prevents the model from erroneously modeling tasks as complete earlier than the "actual"

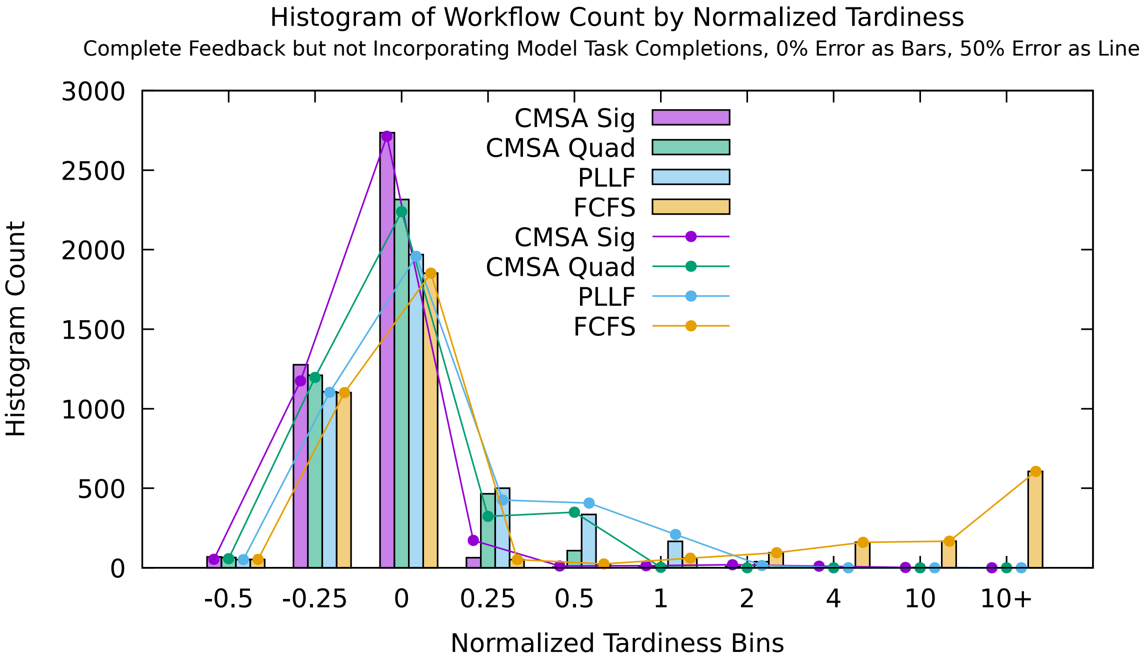

Figure 6.5: Effect on the histogram of workflows completed by normalized tardiness of the presence of "actual" system feedback of task completions in the presence of high error (taken from a uniform distribution and applied only to the CPU work requirement of tasks) for all scheduling algorithms.

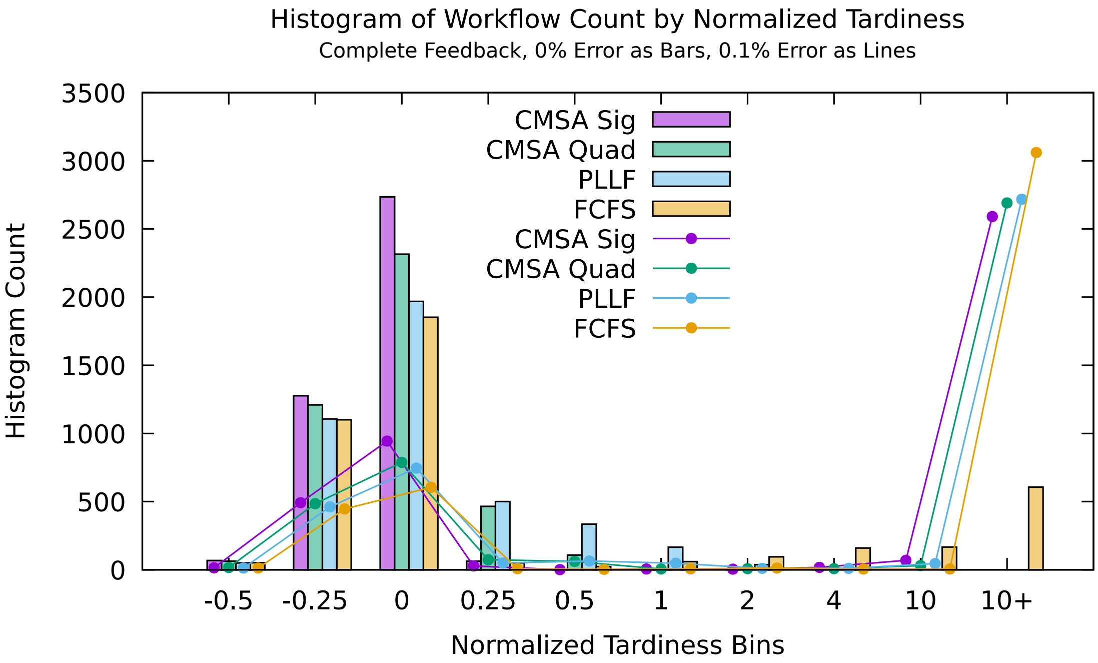

Figure 6.6: Effect on the histogram of workflows completed by normalized tardiness of the presence of "actual" system feedback of task completions, though model task completions are also acted on for scheduling purposes, in the presence of low error (taken from a uniform distribution and applied only to the CPU work requirement of tasks) for all scheduling algorithms.

system. When this happens all scheduling algorithms naturally choose to schedule additional tasks to execute on the machine(s) which the model represents as being less loaded, which leads to the poor performance of the algorithms. To illustate this Figure 6.6 depicts the same histogram result in the presence of feedback and low (up to 0.1% error) but where the model is allowed to model tasks as completed (before the actual task completes, due to the model error). It shows that each scheduling algorithm again has drastically worse performance as in Figure 6.2 despite feedback being completely available because of modeling tasks as completed earlier than the "actual" task completes.

Thus feedback of task completions and not incorporating task completions of the model are influential aspects on the outcome performance of the four scheduling algorithms under study. The next logical question is whether full feedback of task completion times is necessary or whether partial feedback may be sufficient for achieving similar performance as with full feedback. Figure 6.6 suggests that since even full feedback is not enough to counter the ill effect of incorporating model task completions that partial feedback will be of no values (since with partial feedback the model's completion of tasks must be incorporated because feedback of that task's completion on the "actual" system may not be available). Figures 6.7, 6.8, 6.9, and 6.10 show that for FCFS, PLLF, CMSA with a sigmoid cost function, and CMSA with a quadratic cost function, repectively, the introduction of model task completions drastically affects performance and that the level of feedback or loss thereof is of little consequence thereafter.

To a very small degree for PLLF, the higher the level of feedback loss the fewer workflows are completed on time, with more workflows completing 10 times, or more, later than their deadline. For CMSA with either cost function that trend is perhaps (again to a very small degree) opposite with more feedbck loss resulting in fewer workflows completed very late and more completed on time. Nonetheless, when results of any level of feedback loss are compared with results having complete feedback it is clear that all scheduling algorithms are not robust (unable to achieve similar tardiness outcomes) to even small model error. Because any amount of feedback less than 100% requires accepting that tasks are complete when modeled as such in the model system, the same problem of scheduling additional tasks to a machine which has fewer executing tasks in the model than in the "actual" system leading to much lower machine efficiency reoccurs. With the presence of complete feedback, all four scheduling algorithms are robust with respect to model error in the CPU work requirement of tasks up to a high, 50%,

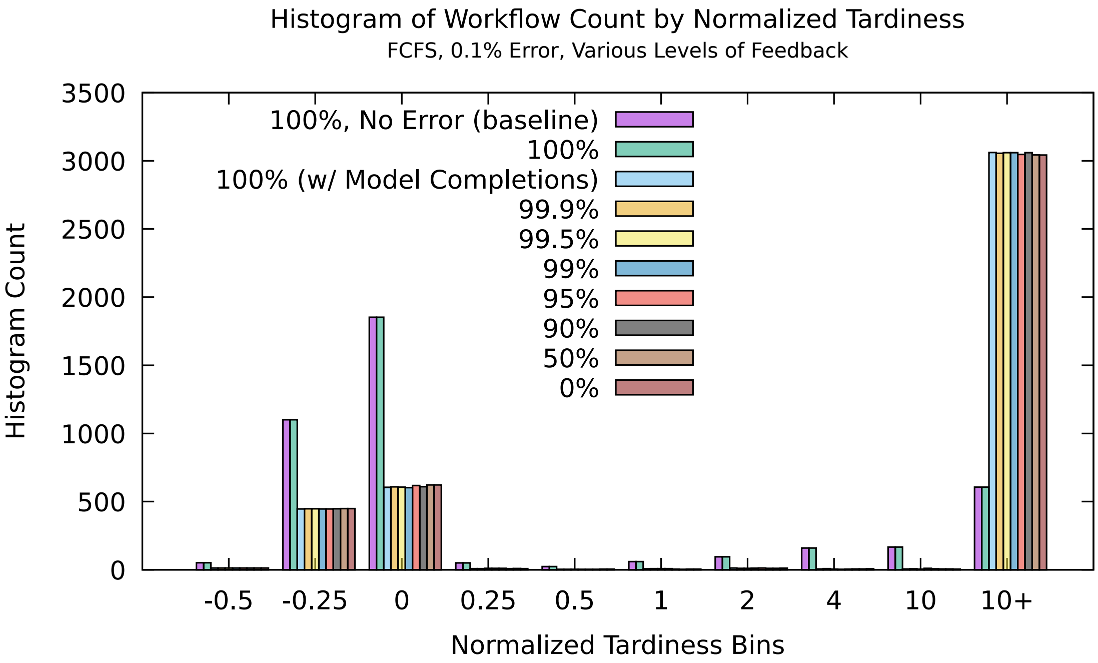

Figure 6.7: Effect on the histogram of workflows completed by normalized tardiness of the level of "actual" system feedback of task completions in the presence of low error (taken from a uniform distribution and applied only to the CPU work requirement of tasks) for the FCFS scheduling algorithm.

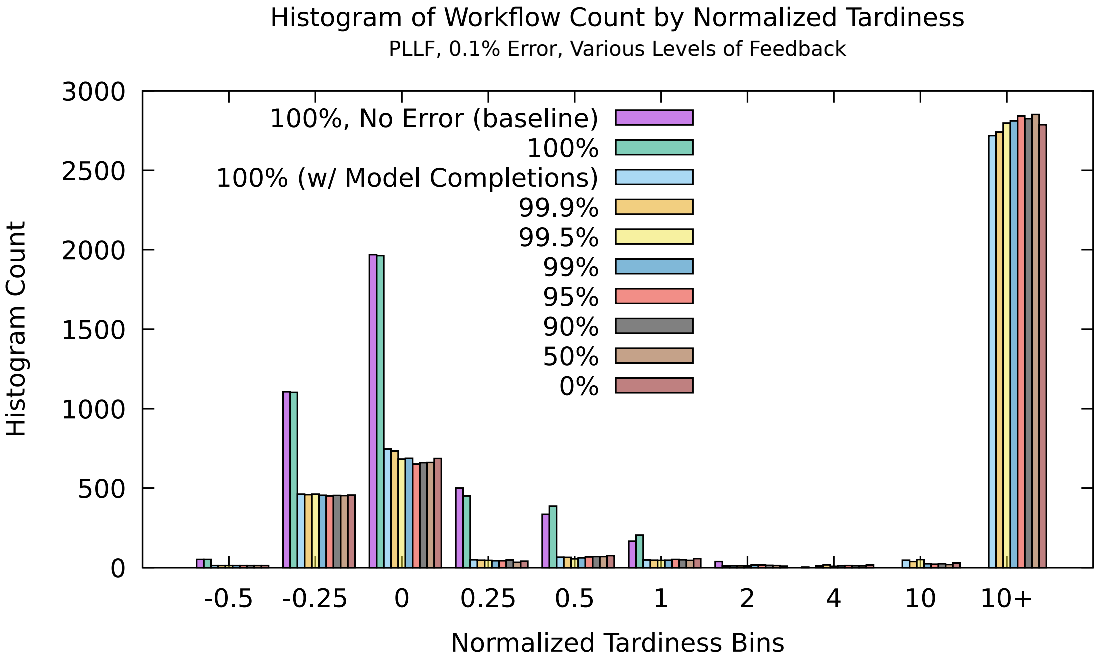

Figure 6.8: Effect on the histogram of workflows completed by normalized tardiness of the level of “actual” system feedback of task completions in the presence of low error (taken from a uniform distribution and applied only to the CPU work requirement of tasks) for the PLLF scheduling algorithm.

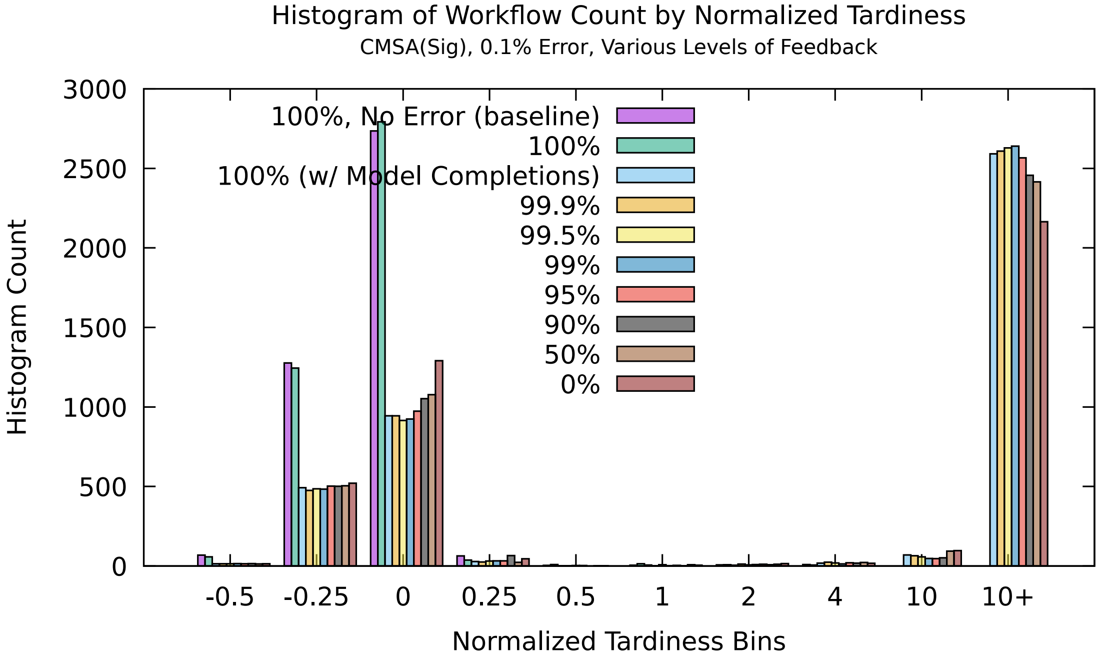

Figure 6.9: Effect on the histogram of workflows completed by normalized tardiness of the level of "actual" system feedback of task completions in the presence of low error (taken from a uniform distribution and applied only to the CPU work requirement of tasks) for the CMSA scheduling algorithm with sigmoid cost function.

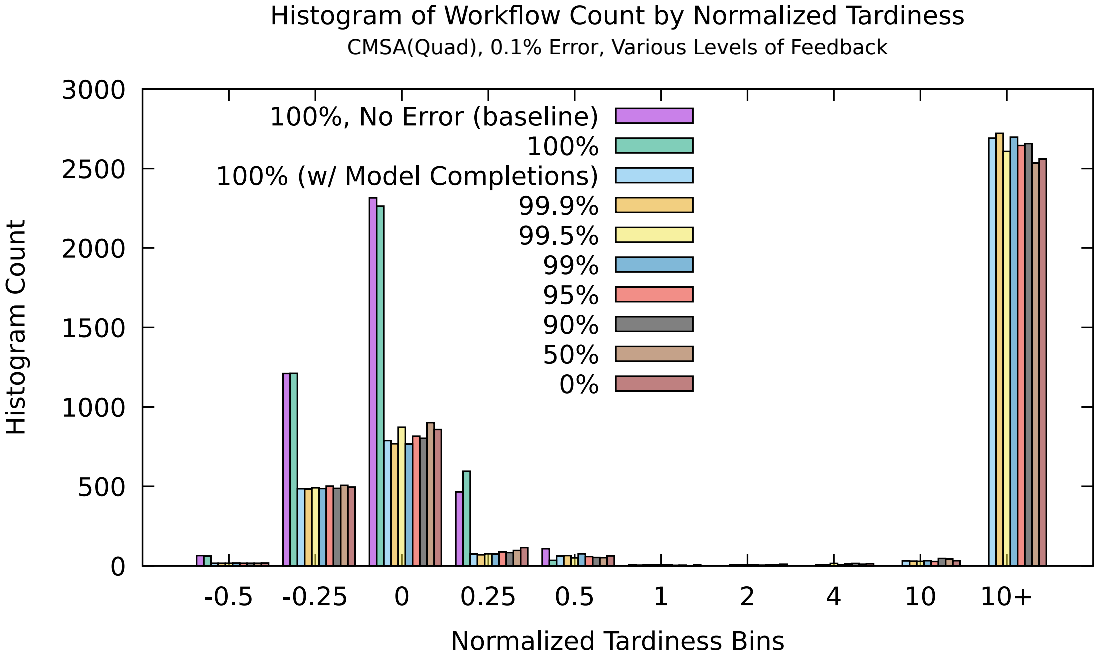

Figure 6.10: Effect on the histogram of workflows completed by normalized tardiness of the level of "actual" system feedback of task completions in the presence of low error (taken from a uniform distribution and applied only to the CPU work requirement of tasks) for the CMSA scheduling algorithm with quadratic cost function.

degree of error.

When error is applied to all model task requirements (CPU work, CPU utilization, and memory utilization) scheduling algorithms become less robust to such error even with the use of complete feedback. Figures 6.11, 6.12, 6.13, and 6.14 depict for FCFS, PLLF, CMSA with sigmoid cost function, and CMSA with quadratic cost function, respectively, the effects of various levels of error applied to all task requirements in the presence of complete feedback (and not incorporating model task completions). All four algorithms are relatively robust to error levels as high as 10% but generally perform noticeably worse with 50% error (though not nearly as poor as having incorporated model task completions even with small error in the model, as depicted in Figure 6.6). FCFS, as an algorithm that relies on no aspect of the workflow/task model for which error is present, is the most robust. However, even FCFS suffers performance loss with high enough error because although the error doesn't affect the order of FCFS prefers to schedule tasks for execution, error does result in model machines appearing to be over or under loaded compared to the "actual" system's machine and lead to scheduling more or fewer tasks for execution concurrently than would be scheduled without the error present. This is why for small enough error bounds (1% or less) the results in Figure 6.11 show FCFS is robust: achieves an equivalent outcome as with an error-free model.

6.4 Effect of Error Biasing

As illustrated previously although feedback from the "actual" system is the primary method by which scheduling algorithms can be made robust to model error (by keeping the model system from erroneously modeling tasks as completed

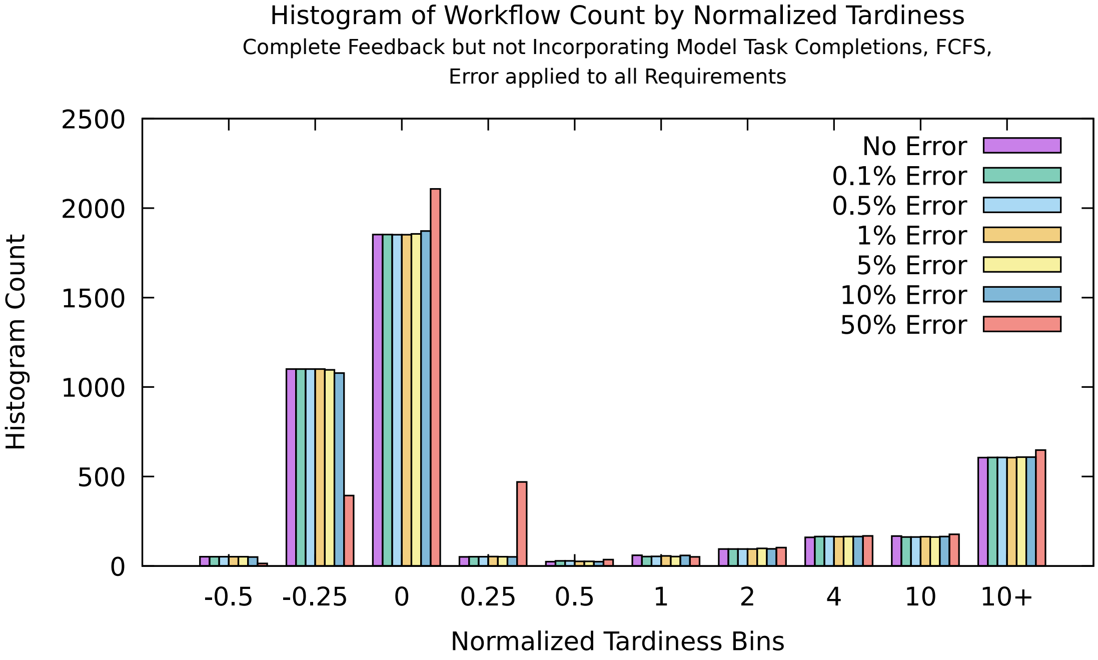

Figure 6.11: Effect on the histogram of workflows completed by normalized tardiness of the level of error with complete "actual" system feedback of task completions and not incorporating model task completions (error taken from a uniform distribution and applied all task requirements) for the FCFS scheduling algorithm.

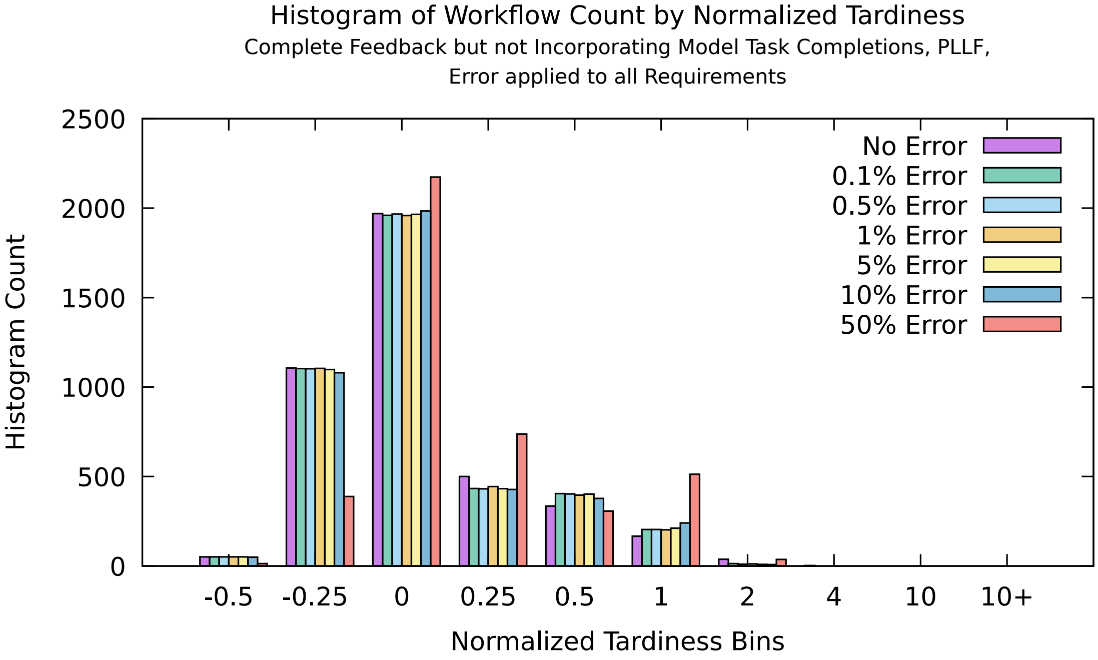

Figure 6.12: Effect on the histogram of workflows completed by normalized tardiness of the level of error with complete "actual" system feedback of task completions and not incorporating model task completions (error taken from a uniform distribution and applied all task requirements) for the PLLF scheduling algorithm.

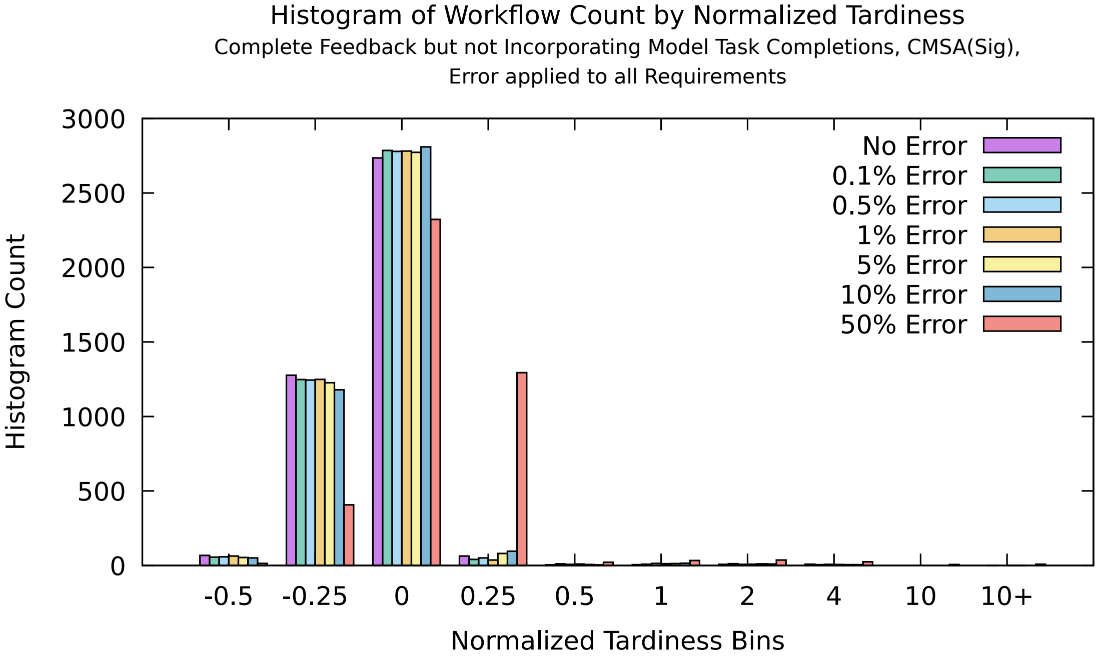

Figure 6.13: Effect on the histogram of workflows completed by normalized tardiness of the level of error with complete "actual" system feedback of task completions and not incorporating model task completions (error taken from a uniform distribution and applied all task requirements) for the CMSA scheduling algorithm with a sigmoid cost function.

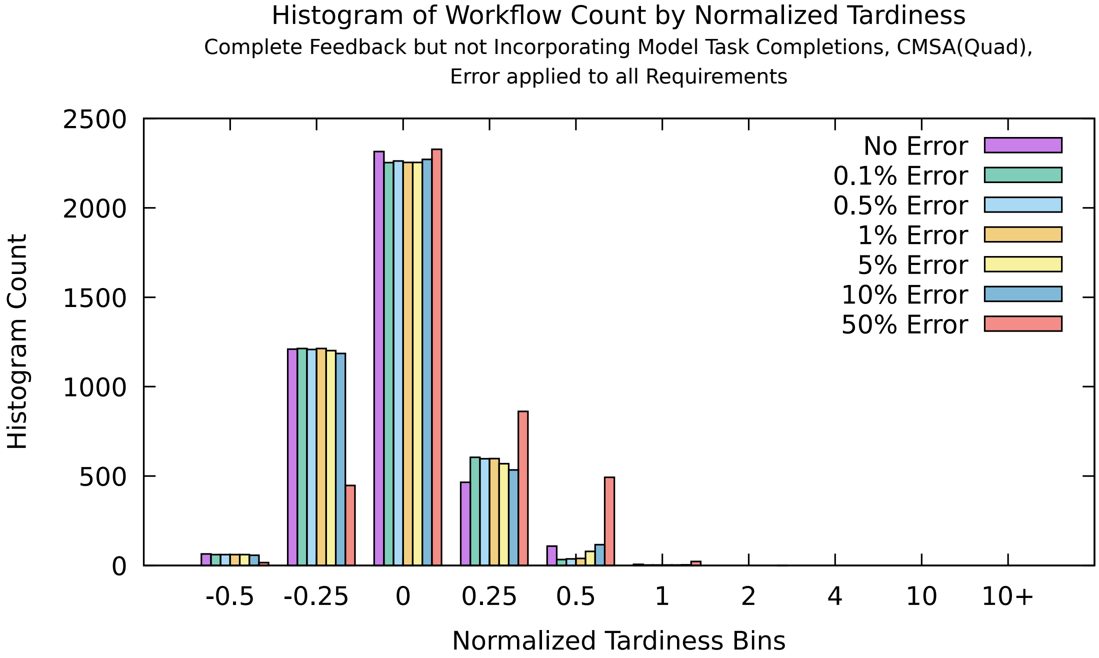

Figure 6.14: Effect on the histogram of workflows completed by normalized tardiness of the level of error with complete "actual" system feedback of task completions and not incorporating model task completions (error taken from a uniform distribution and applied all task requirements) for the CMSA scheduling algorithm with a quadratic cost function.

early) biasing the error inherent in the model may also achieve a similar effect. As discussed in Section 5.3 because the exact nature and magnitude of the error is likely unknown, model task requirements must be biased using, ideally, some notion of what the maximum expected error value or percentage is which may underestimate the true value of the requirement. Also discussed were three separate biasing strategies: first, the constant bias, second, the proportionate bias (see Eq. 5.2), and finally the simple bias (see Eq. 5.3).

Figures 6.15, 6.16, 6.17, and 6.18 depict the FCFS, PLLF, CMSA with sigmoid cost function, and CMSA with quadratic cost function algorithms, respectively, with the highest simulated amount of error (up to 50%, taken from a uniform number distribution) for various levels of "actual" system feedback (or loss of such feedback) with a constant bias applied to the "model" system task requirements. Generally, for all four algorithms the impact of incomplete feedback as low as 99% is relatively very small compared with having full feedback.

In Figures 6.15 and 6.16 the FCFS and PLLF algorithms actually appears to perform better (more workflows completed much earlier than their deadline) with a model having error but biased vs. the performance of a model with no error given 90% or more feedback. This is demonstrated by the first, purple bar in the histogram having a lower value for negative normalized tardinesses and a higher value for positive normalized tardinesses. At feedback levels lower than 90% (i.e., 50% and 0%, being the only two feedback levels simulated which were less than 90%) even the biased model error performs very poorly, with far fewer workflows completed on time and many more completed many times later than their deadline. For FCFS this comes in the form of workflows complete ten or more times later than their deadline. For PLLF this comes in the form of workflows completed between one and four times later than their deadline.

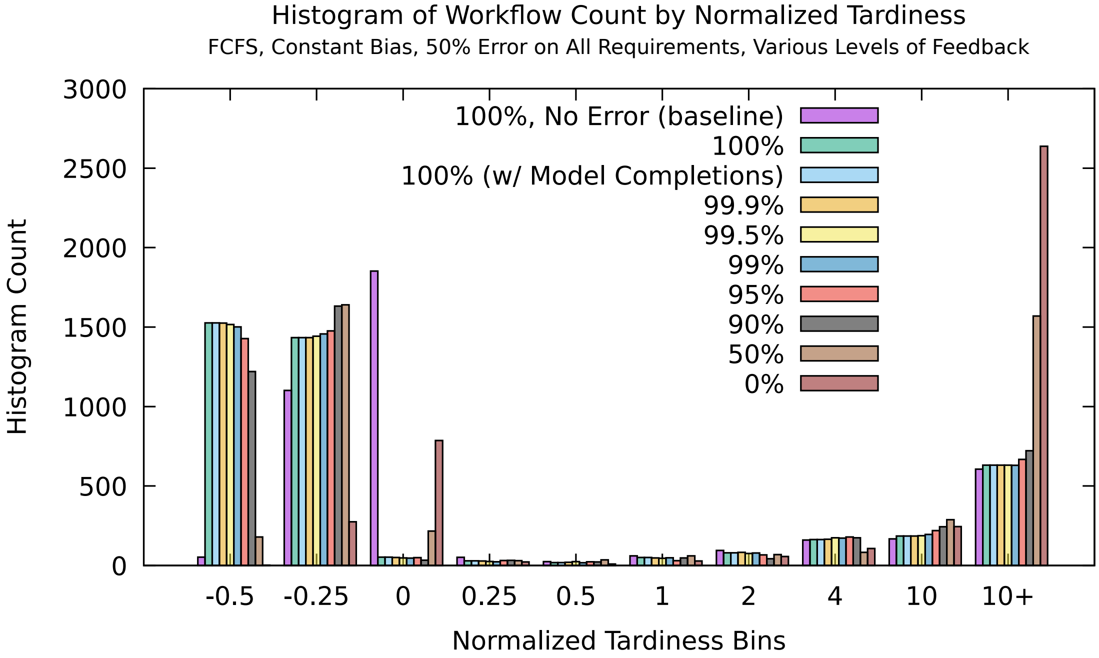

Figure 6.15: Effect on the histogram of workflows completed by normalized tardiness of the level of “actual” system feedback in the presence of high (50%) error (taken from uniform distribution) on all task requirements using the constant bias strategy (constant value, C, perfectly matches e which is 50%) for the FCFS scheduling algorithm.

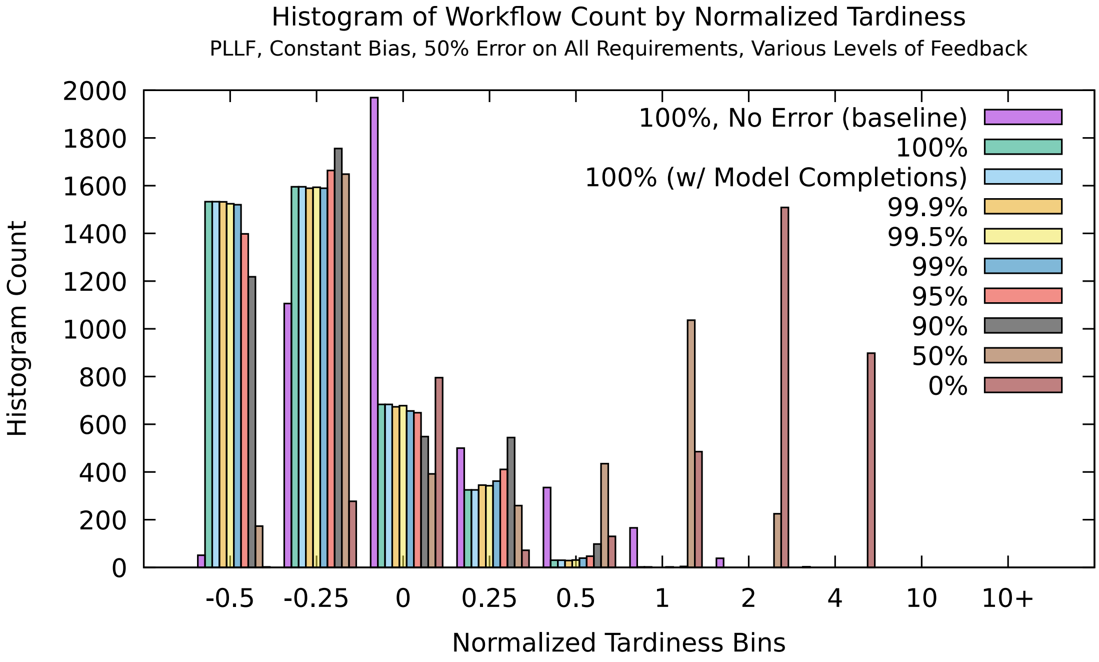

Figure 6.16: Effect on the histogram of workflows completed by normalized tardiness of the level of "actual" system feedback in the presence of high (50%) error (taken from uniform distribution) on all task requirements using the constant bias strategy (constant value, C, perfectly matches e which is 50%) for the PLLF scheduling algorithm.

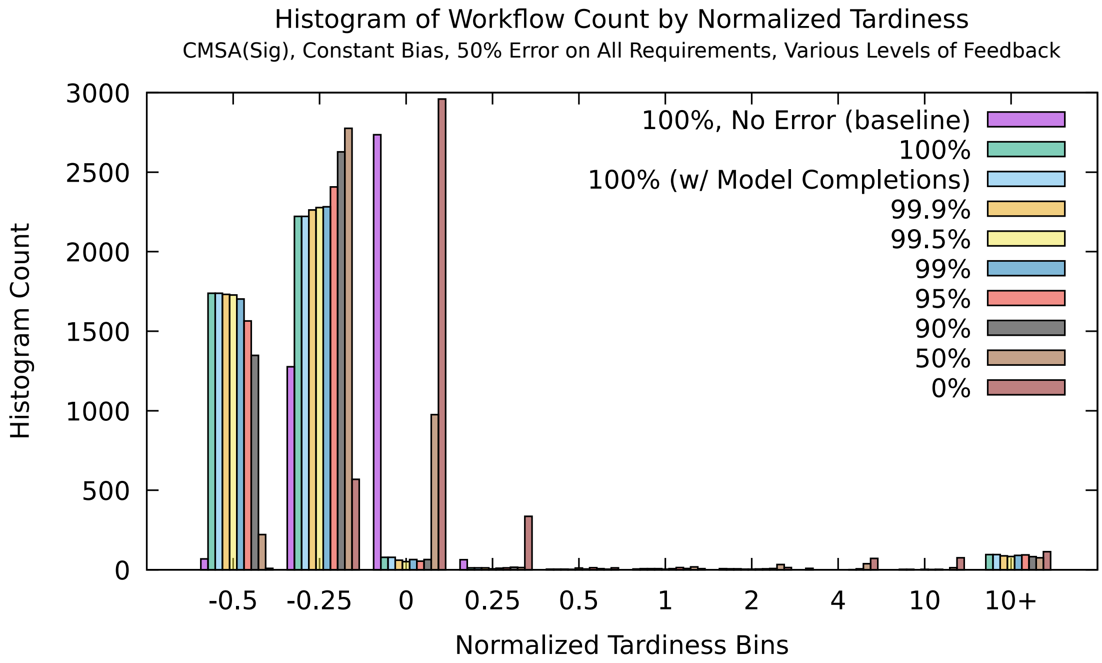

Figure 6.17: Effect on the histogram of workflows completed by normalized tardiness of the level of "actual" system feedback in the presence of high (50%) error (taken from uniform distribution) on all task requirements using the constant bias strategy (constant value, C, perfectly matches e which is 50%) for the CMSA scheduling algorithm with sigmoid cost function.

In Figure 6.17 the CMSA algorithm with a sigmoid cost function demonstrates a very different outcome. Although performance with high levels of feedback (99% and above) appear to be roughly equal and do complete far more workflows ahead of their deadline than the baseline case with a error-free model, that performance comes at the cost of having a non-trivial amount (about 100 of the 4,354 workflows) completing 10 times or more later than their deadline. However, unlike PLLF, though the lowest levels of feedback don't appear to complete workflows nearly as much ahead of their deadline as with cases of higher feedback levels, they still complete mostly by the time of the deadline (normalized tardiness between -0.25 and 0).

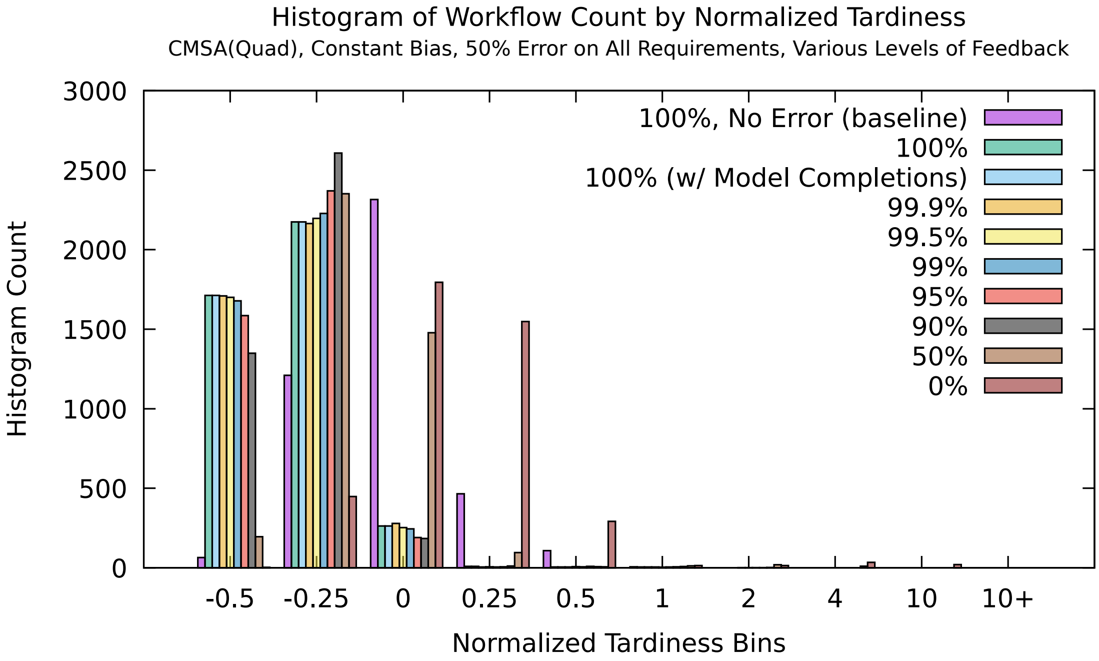

Figure 6.18: Effect on the histogram of workflows completed by normalized tardiness of the level of "actual" system feedback in the presence of high (50%) error (taken from uniform distribution) on all task requirements using the constant bias strategy (constant value, C, perfectly matches e which is 50%) for the CMSA scheduling algorithm with quadratic cost function.

In Figure 6.18 the CMSA algorithm with a quadratic cost function demonstrates, as with PLLF and CMSA with sigmoid, that far more workflows are completed much earlier than their deadline than with an error-free model when using a constant bias. Unlike CMSA with the sigmoid cost function, however, there are no workflows completed 10 or more times later than their deadline, and only a few workflows completed 1 to 10 times later than their deadline and only for the lowest two levels of feedback. Thus the CMSA algorithm when using a quadratic cost function appears to be robust to model error as high as 50% with feedback levels as low as 50% with the use of a constant bias strategy.

Unfortunately, the simple bias strategy performs very poorly for all algorithms

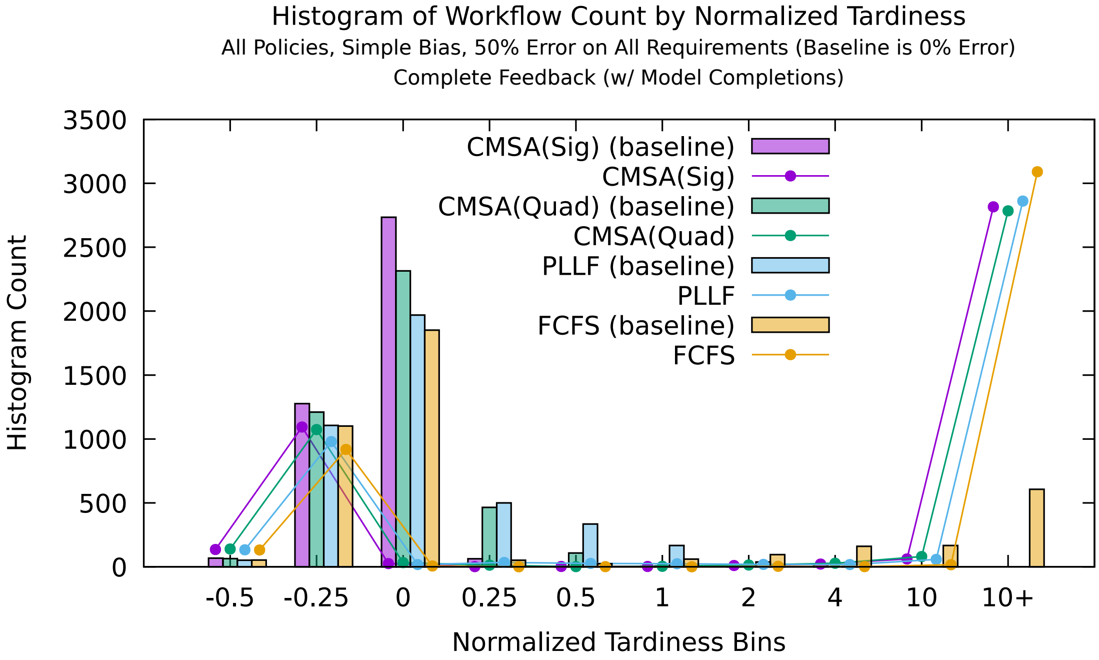

Figure 6.19: Effect on the histogram of workflows completed by normalized tardiness of the use of the simple bias strategy ($\hat{e}$ perfectly matches e which is 50%) in the presence of high (50%) error (taken from uniform distribution) on all task requirements for the FCFS, PLLF, and CMSA scheduling algorithms.

in the presence of high (50%) error in task requirements regardless of how much feedback from the "actual" system is present. Figure 6.19 shows that for FCFS, PLLF, and CMSA (both with sigmoid and quadratic cost functions) comparing to a baseline of no error, the high (50%) error with simple biasing performs far worse, completes many fewer workflows early and many more workflows 10 or more times later than their deadline. Because the simple bias strategy does not completely overcome the possibility that error in the model can result in modeling tasks as complete before they are in the "actual" system the problem of scheduling too many tasks on a machine resulting in poor efficiency (and further deviation between the modeled and actual finish time of future tasks) is still present.

Figure 6.20 demonstrates a similar, though less dramatic, result for the proportionate bias strategy. In it the number of workflows completed early or on time in the presence of high error with a proportionate bias is substantially less than with an error-free model. However, unlike the simple bias strategy only the FCFS and CMSA algorithm using a sigmoid cost function complete workflows 10 or more times later than their deadline. For PLLF and CMSA with a quadratic cost function the worst-case completion is kept under 10 times later than the deadline, with most workflows completing only 4 or less times later than the deadline. Although the proportionate bias strategy with a suitably large enough $\hat{e}$ (as was the case in Figure 6.20) prevents underestimating the completion time of tasks in the model, that guarantee comes at the cost of "stretching" the distribution of the biased task requirement in the model, resulting in overestimates that can be wildly inaccurate when compensating for high error (see results in Figure 5.5 from Section 5.3 and its explanation).

Where the simple and proportionate bias strategies yield better results (closer to that of an error-free model) is in the presense of smaller error. If the error present in the model is bounded by 5% as opposed to 50% as in prior results, the performance of FCFS using a model biased with the simple bias strategy (see Figure 6.22) is nearly as good as using the constant bias strategy (see Figure 6.21).

In both the use of the constant bias strategy (Figure 6.24) and simple bias strategy (Figure 6.25) the FCFS algorithm for feedback levels of at least 50% is able to complete more workflows much earlier than their deadline, and generally fewer workflows later than their deadline than in the case of an error-free model. In the test scenarios presented in the figures only for 0% feedback does FCFS complete fewer workflows early and more workflow late than the error-free model.

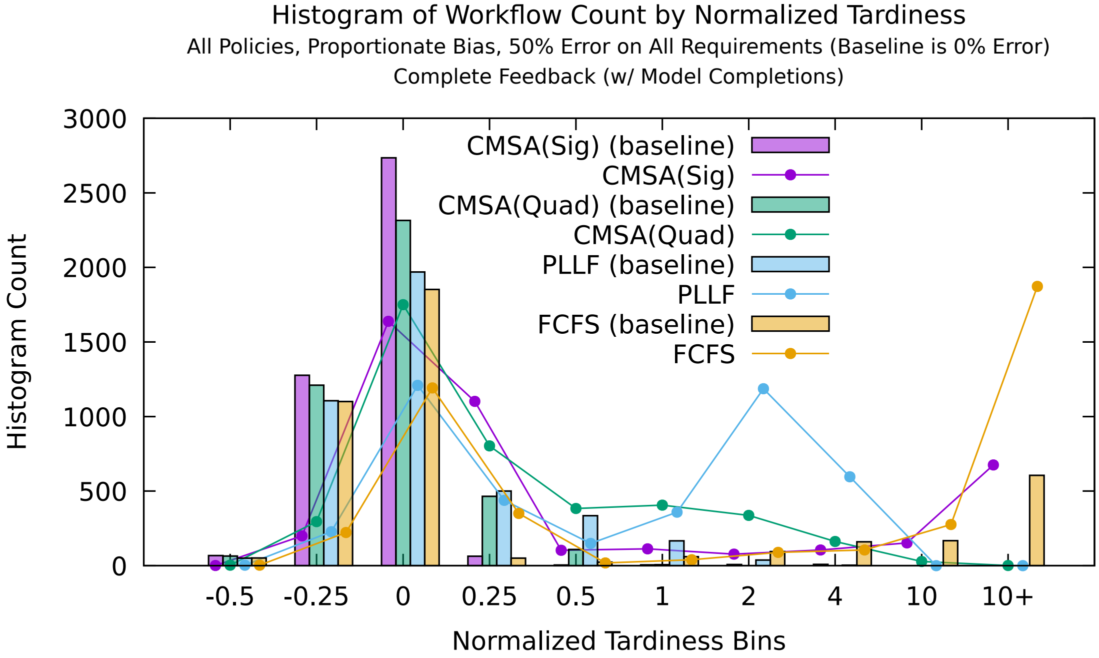

Figure 6.20: Effect on the histogram of workflows completed by normalized tardiness of the use of the proportionate bias strategy ($\hat{e}$ perfectly matches e which is 50%) in the presence of high (50%) error (taken from uniform distribution) on all task requirements for the FCFS, PLLF, and CMSA scheduling algorithms.

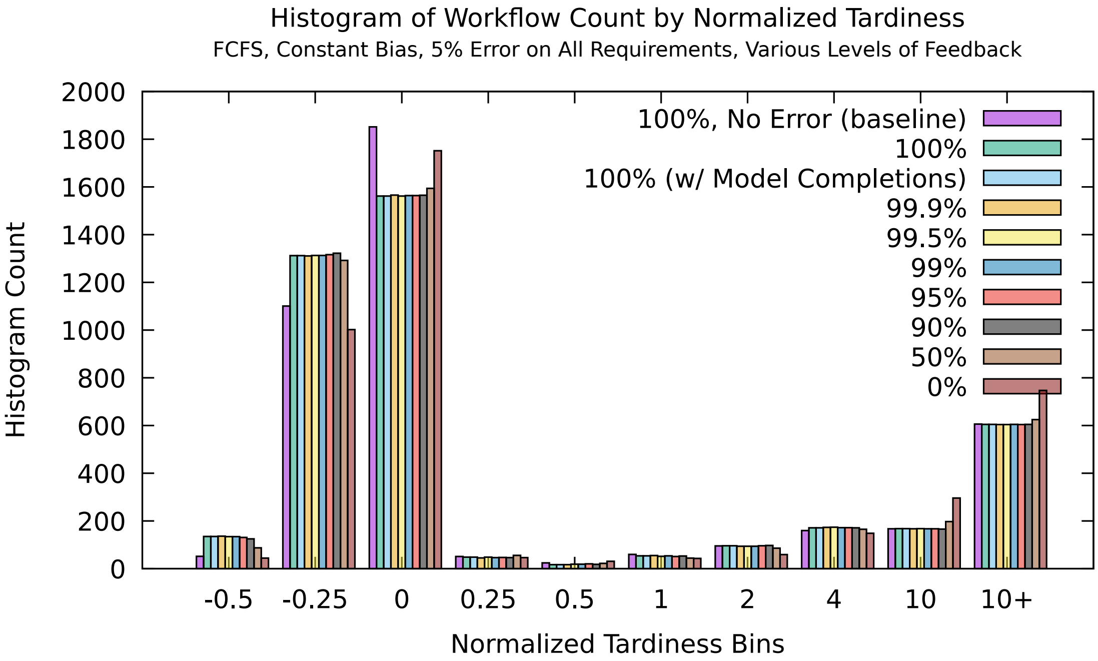

Figure 6.21: Effect on the histogram of workflows completed by normalized tardiness of the use of the constant bias strategy (with ideal $C = e = 50\%$) for FCFS in the presence of medium (5%) error (taken from uniform distribution) on all task requirements.

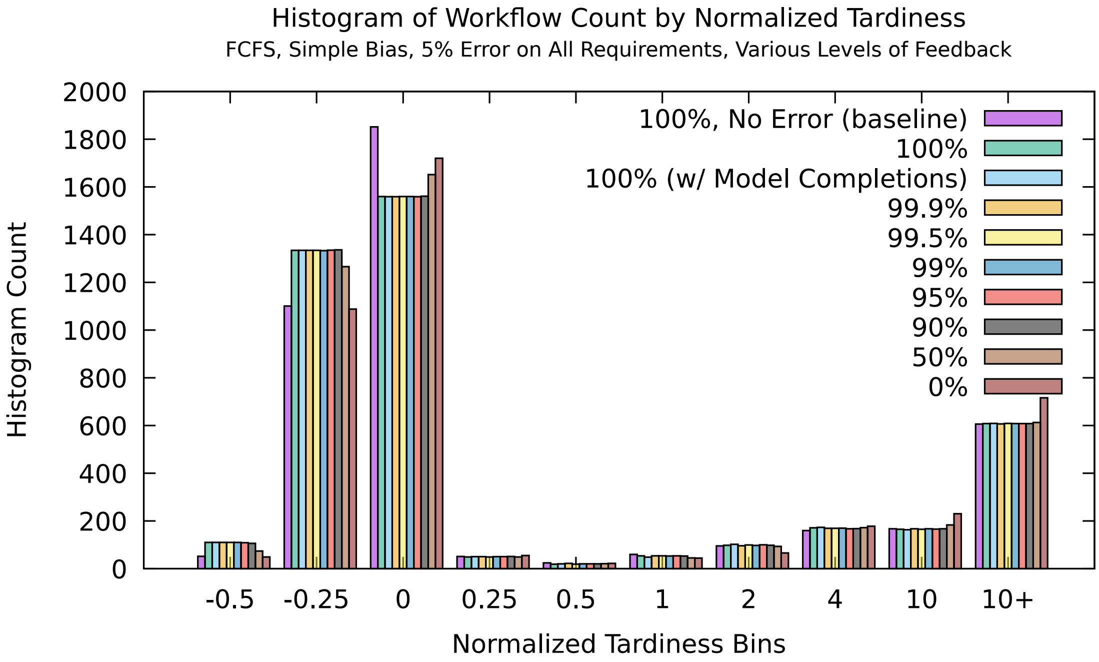

Figure 6.22: Effect on the histogram of workflows completed by normalized tardiness of the use of the simple bias strategy (with ideal $\hat{e} = e = 50\%$) for FCFS in the presence of medium (5%) error (taken from uniform distribution) on all task requirements.

Figure 6.23 shows the performance of FCFS in the same 5% bounded-error scenario but for the proportionate bias strategy. In it the performance at most feedback levels up to 50% is nearly the same as the error-free model, but generally any loss of feedback results in some reduction in the number of workflows completed early and increase in workflows completed late. Thus, though the simple bias strategy may not (with $\hat{e} = e$) prevent underestimating task requirements in all circumstances, with a smaller bounded error, the probability that a few underestimates (vs. the more numerous overestimates) results in the over-allocation of tasks to a machine to the extent of causing severe deviation between the model machine at the "actual" machine (see Figure 6.3) appears to be negligible, and the simple bias strategy is able to achieve the same level of performance (closely match the outcome of an error-free model) as the constant bias strategy. Whereas the "stretching" effect of the proportionate bias strategy (as the trade-off to never underestimating task requirements for $\hat{e} = e$) appears to have a more impactful negative effect on performance.

The results for the PLLF and CMSA algorithm (for both sigmoid and quadratic cost functions) in the presence of 5% bounded error in the model for the three bias strategies mimics the results of FCFS: simple bias strategy achieves nearly identical results as the constant bias strategy, with the proportionate bias strategy performing somewhat worse for feedback levels as low as about 50%. Figures 6.24 - 6.32 depict all these results. The one difference is that the CMSA algorithm with a sigmoid cost function and the simple bias strategy seems to maintain performance equal to the baseline of an error-free model even with no feedback from the "actual" system except for a very few workflows (18 vs. 7) completed between one and two times later than their deadline, and even 2 and 1 workflows completed between four and ten times later than their deadline, and more than

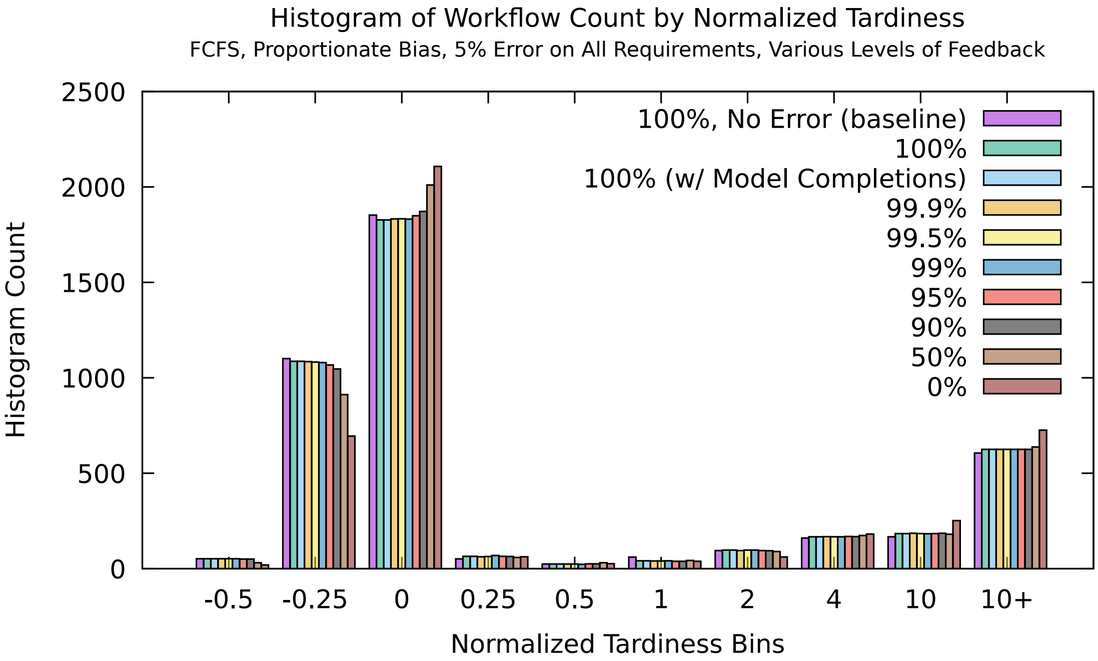

Figure 6.23: Effect on the histogram of workflows completed by normalized tardiness of the use of the proportionate bias strategy (with ideal $C = e = 50\%$) for FCFS in the presence of medium (5%) error (taken from uniform distribution) on all task requirements.

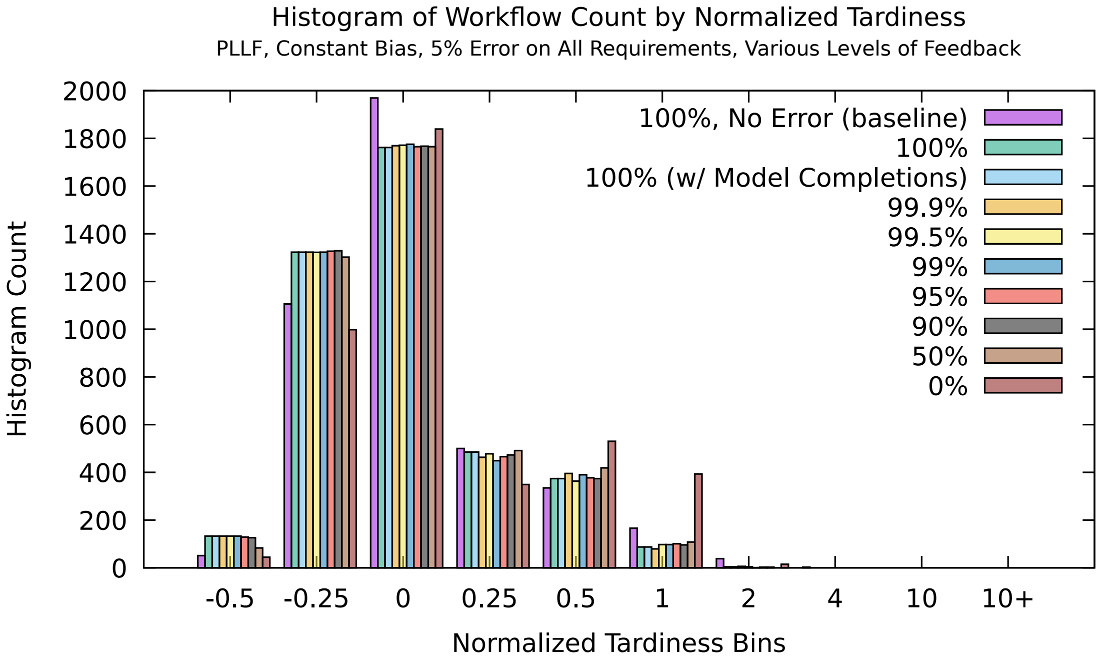

Figure 6.24: Effect on the histogram of workflows completed by normalized tardiness of the use of the constant bias strategy (with ideal $C = e = 50\%$) for PLLF in the presence of medium (5%) error (taken from uniform distribution) on all task requirements.

ten times later than its deadline, respectively.

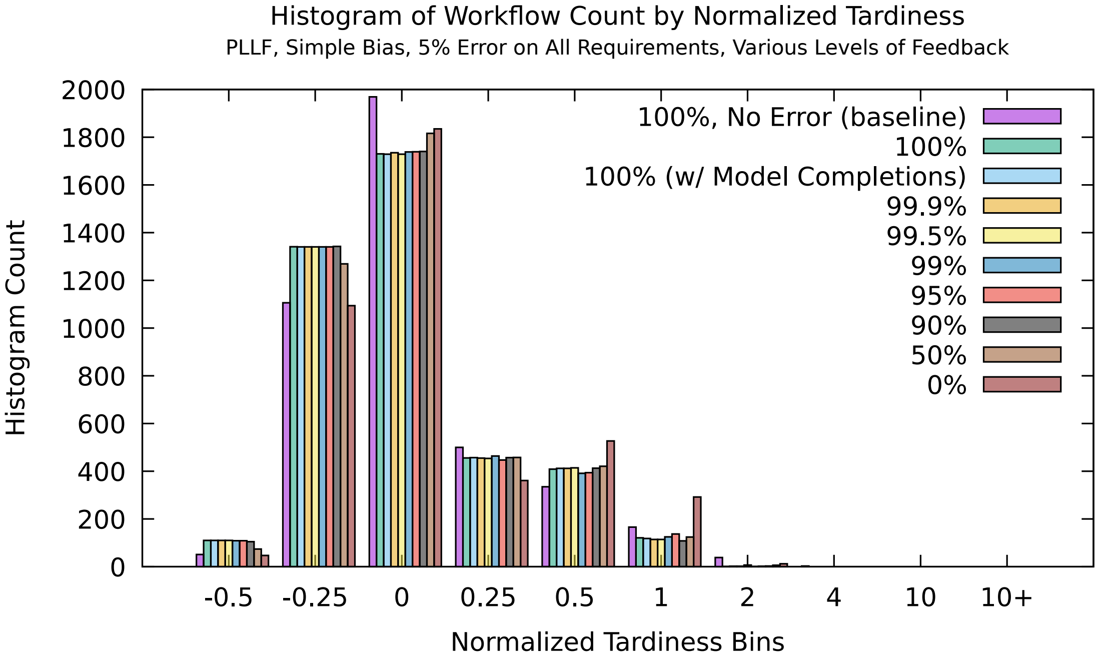

Figure 6.25: Effect on the histogram of workflows completed by normalized tardiness of the use of the simple bias strategy (with ideal $\hat{e} = e = 50\%$) for PLLF in the presence of medium (5%) error (taken from uniform distribution) on all task requirements.

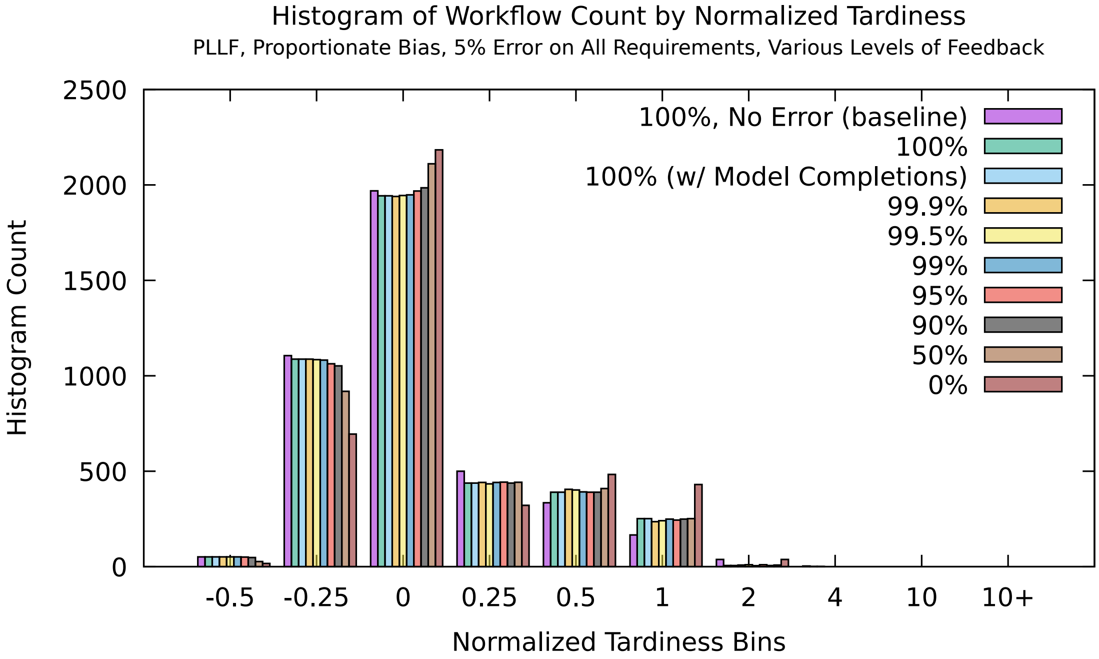

Figure 6.26: Effect on the histogram of workflows completed by normalized tardiness of the use of the proportionate bias strategy (with ideal $C = e = 50\%$) for PLLF in the presence of medium (5%) error (taken from uniform distribution) on all task requirements.

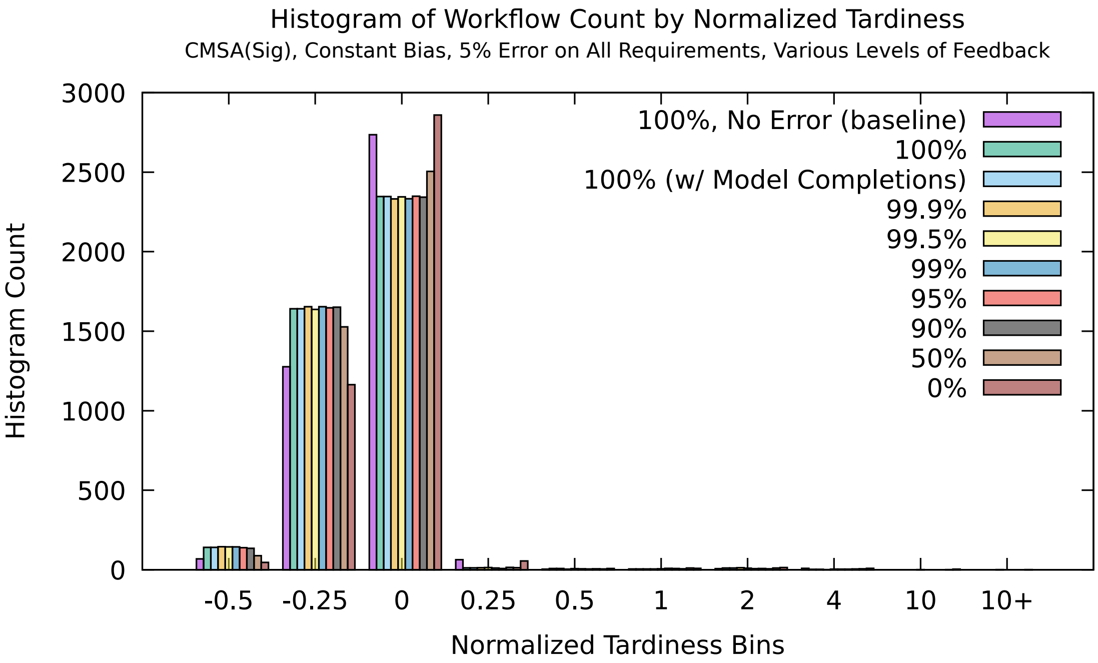

Figure 6.27: Effect on the histogram of workflows completed by normalized tardiness of the use of the constant bias strategy (with ideal $C = e = 50\%$) for CMSA (with sigmoid cost function) in the presence of medium (5%) error (taken from uniform distribution) on all task requirements.

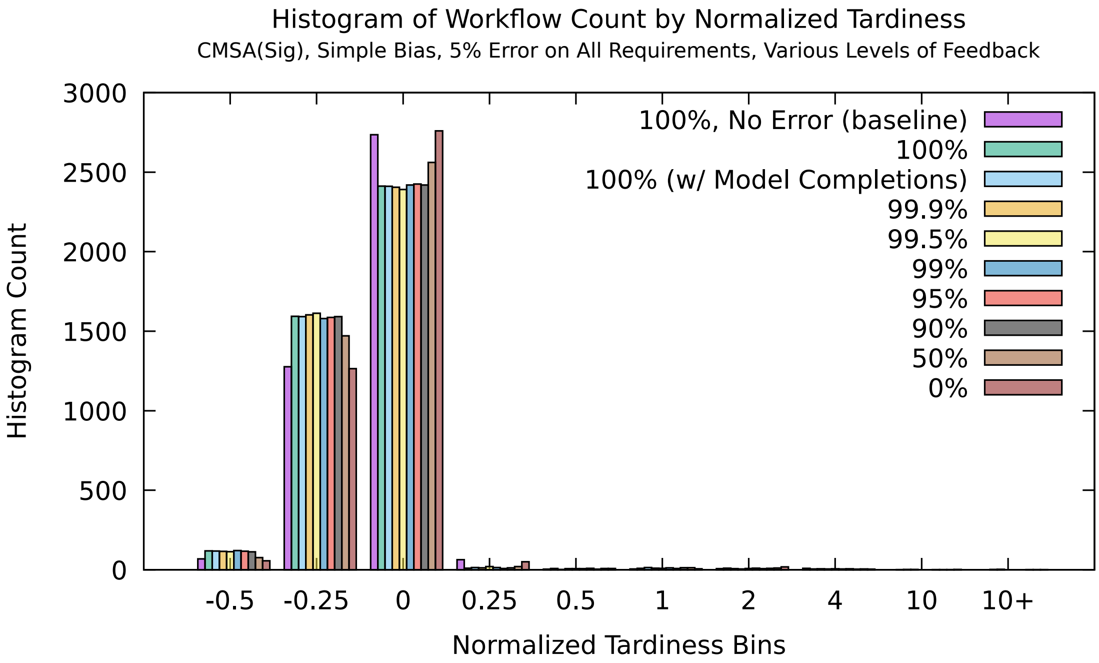

Figure 6.28: Effect on the histogram of workflows completed by normalized tardiness of the use of the simple bias strategy (with ideal $\hat{e} = e = 50\%$) for CMSA (with sigmoid cost function) in the presence of medium (5%) error (taken from uniform distribution) on all task requirements.

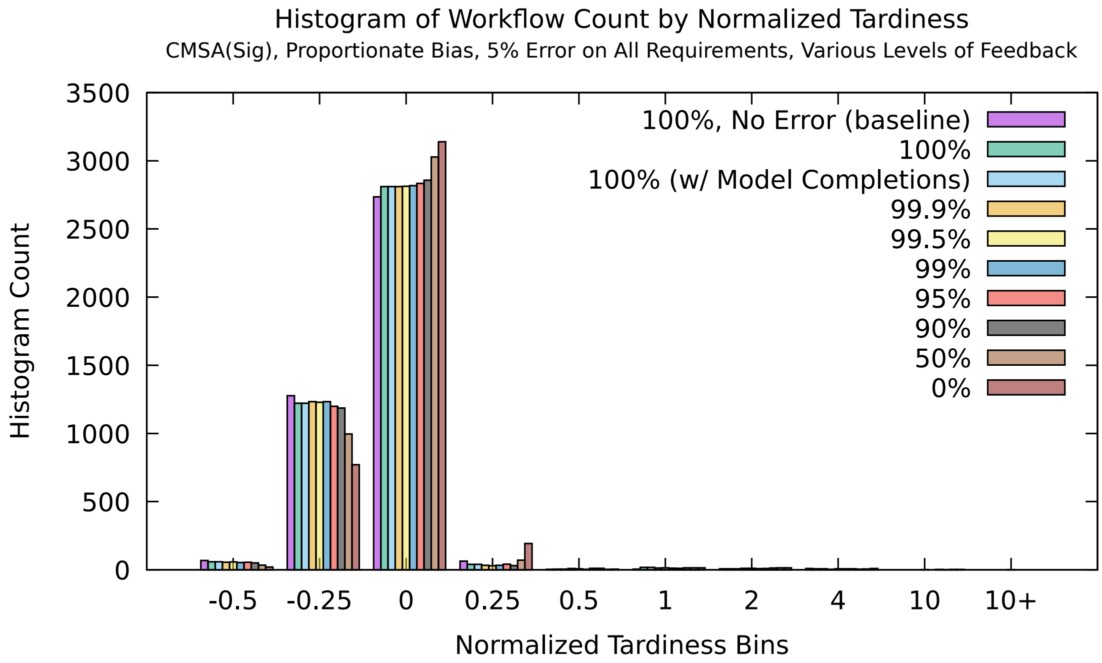

Figure 6.29: Effect on the histogram of workflows completed by normalized tardiness of the use of the proportionate bias strategy (with ideal $\hat{e} = e = 50\%$) for CMSA (with sigmoid cost function) in the presence of medium (5%) error (taken from uniform distribution) on all task requirements.

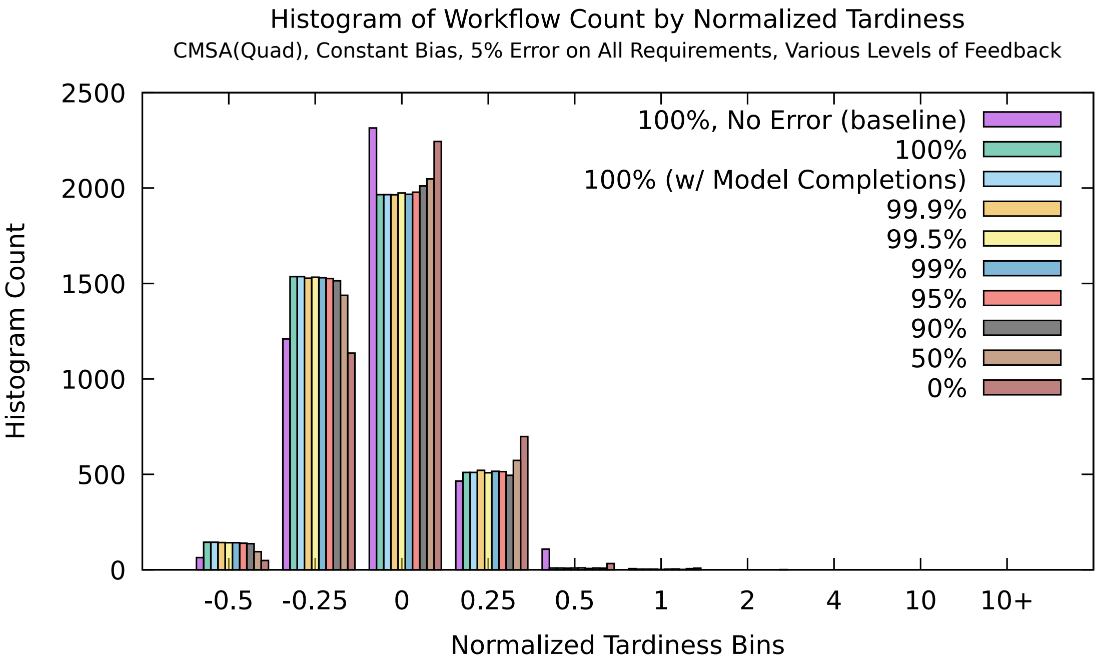

Figure 6.30: Effect on the histogram of workflows completed by normalized tardiness of the use of the constant bias strategy (with ideal $C = e = 50\%$) for CMSA (with quadratic cost function) in the presence of medium (5%) error (taken from uniform distribution) on all task requirements.

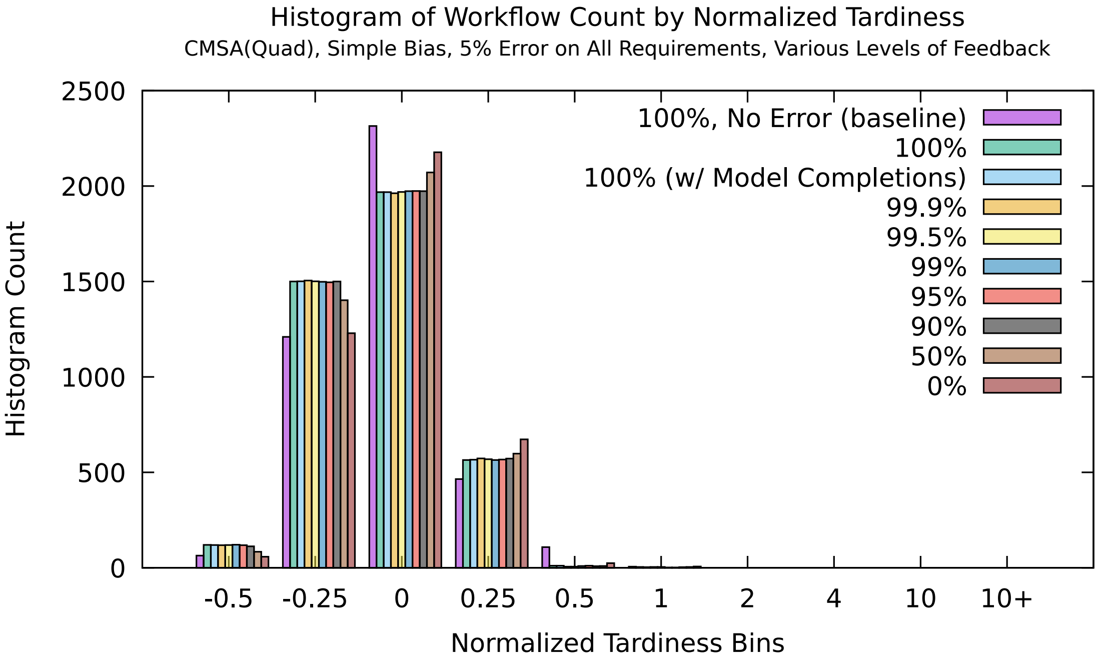

Figure 6.31: Effect on the histogram of workflows completed by normalized tardiness of the use of the simple bias strategy (with ideal $\hat{e} = e = 50\%$) for CMSA (with quadratic cost function) in the presence of medium (5%) error (taken from uniform distribution) on all task requirements.

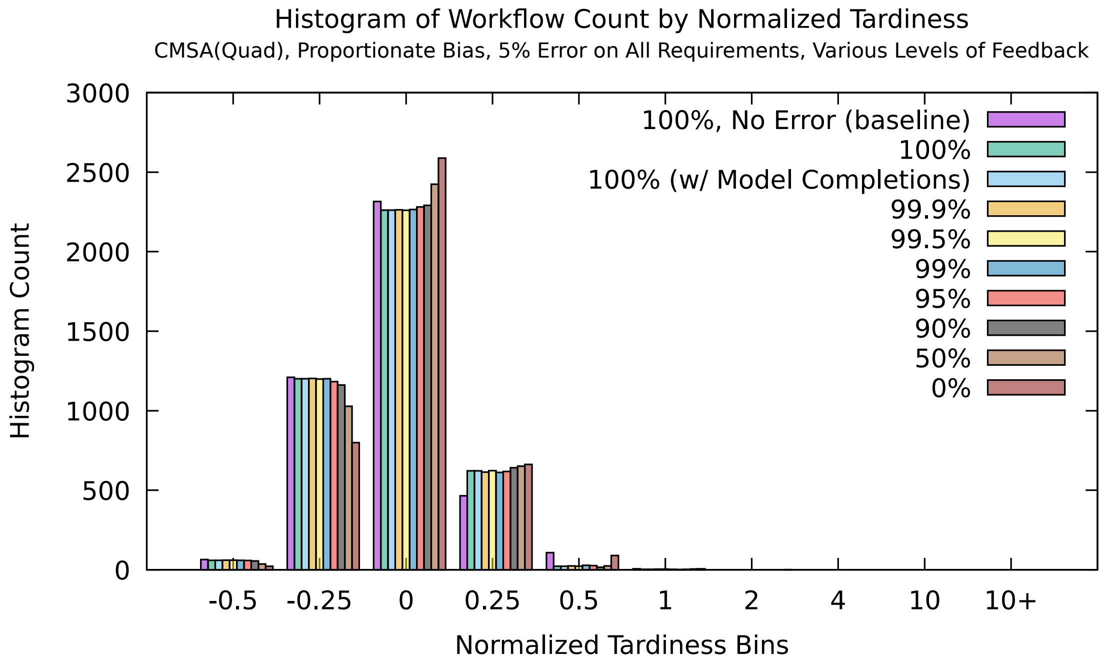

Figure 6.32: Effect on the histogram of workflows completed by normalized tardiness of the use of the proportionate bias strategy (with ideal $\hat{e} = e = 50\%$) for CMSA (with quadratic cost function) in the presence of medium (5%) error (taken from uniform distribution) on all task requirements.

Chapter 7

Conclusions

In this research the impact of model error is considered for scheduling of complex work loads of tasks on resources of machines in a distributed system. Robustness, defined as the degree to which similar performance can be achieved despite the presence of error in the model, is measured as the amount of workflows completed at various proportions relative to their deadline (both earlier than the deadline and later) and is presented visually in histograms.

Through simulated studies of a modeled 24-hour period of system processing the extensive numerical studies reveal the primary factor for achieving robustness lies in the use of feedback from the "actual" system to correct the model. Thus, when tasks are completed, feedback prevents the model, because of error, from declaring tasks as completed early and thus preventing algorithms from scheduling additional work to begin execution. Without such feedback completely available in simulations even the smallest amount of error which could underestimate task requirements resulted in very poor outcomes (drastically fewer workflows completed before or by their deadline and more workflows completed many times later than their deadline). For all four scheduling algorithms studied they exhibited poor robustness to any amount of model error which could underestimate task requirements where “actual” system feedback regarding task completions was not completely available.

In order to increase robustness of scheduling algorithms to model error when “actual” system feedback was not fully available, biasing strategies were employed to prevent or significantly reduce the probability that model error would underestimate task requirements. Three such bias strategies were simulated and demonstrated that robustness to error could be mostly restored despite task completion feedback not being completely available. The first bias strategy, requiring the most a priori knowledge of the nature of task requirements and the bounds of the error present in the model, called the constant bias strategy achieved the best results demonstrating robustness to even the highest simulated amounts of error achievable for all scheduling algorithms. The other bias strategies required less knowledge about the bounds of the error but also were unable to achieve as good of robustness, but were still suitable for lower levels of error in the model. In all cases, the presence of at least some feedback of task completions from the “actual” system was shown to be a critical component of achieving robustness to model error regardless of any of the bias strategies.

7.1 Future Research Ideas

Given the importance of feedback in order to correct the model platform it would be beneficial in future research to consider various types of feedback. In this research only feedback of task completion events, whether all such events or only a random sampling, was considered. Other possible types of feedback include periodic polling or sampling of the actual platform rather than task-based event-driven feedback. This periodic sampling feedback could consider whether to measure which tasks are currently executing which would function similarly to the task completion feedback in this research allowing corrections to the model when it models tasks as completing earlier than on the actual platform. Alternatively, periodic sampling could measure (or estimate) machine efficiency which would function to prevent the model from over-estimating machine efficiency while the actual machines' efficiency is very poor.

Separate from the type and use of feedback another interesting possibility for future research is the inclusion of preemptive scheduling. Preemption in scheduling is often used in Operating System-level scheduling of processes but could be equally useful in distributed system scheduling when a higher priority task arrives in the scheduling queue than some currently running task. Typically in Operating System-level scheduling of processes preemption generally freezes whatever process was executing in the state it is in at the time preemption discontinues its execution. This results in that process resuming where it left off when, at a future time, it is scheduled to execute once again. In distributed system processing a task may be unable to save and resume its state, especially given that the machine it may be executed on in the future could be different than the machine where it previously executed. This would mean that the scheduling decision to preempt one task must take into account the cost of losing work. If the distributed system does support saving task state and resuming it at a later time (or the same or any machine) then the scheduling decision to preempt would still have to take into account the cost (in time or resource usage) for saving a preempted task's state.

Bibliography

[1] Ehab Nabiel Alkhanak, Sai Peck Lee, and Saif Ur Rehman Khan. Cost-aware challenges for workflow scheduling approaches in cloud computing environments: Taxonomy and opportunities. *Future Generation Computer Systems*, 50:3–21, 2015.

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[3] Felice Balarin, Luciano Lavagno, Praveen Murthy, Alberto Sangiovanni-Vincentelli, CD Systems, et al. Scheduling for embedded real-time systems. *IEEE Design & Test of Computers*, 15(1):71–82, 1998.

[4] Marta Beltrán, Antonio Guzmán, and Jose L Bosque. A new cpu availability prediction model for time-shared systems. *IEEE Transactions on Computers*, 57(7):865–875, 2008.

[5] Louis-Claude Canon and Emmanuel Jeannot. Evaluation and optimization of the robustness of dag schedules in heterogeneous environments. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):532–546, 2010.

[6] Michael L. Dertouzos and Aloysius K. Mok. Multiprocessor online scheduling of hard-real-time tasks. *IEEE Transactions on software engineering*, 15(12):1497–1506, 1989.

[7] Dmytro Dyachuk and Ralph Deters. Using sla context to ensure quality of service for composite services. In *Pervasive Services, IEEE International Conference on*, pages 64–67. IEEE, 2007.

[8] Yihong Gao, Huadong Ma, Haitao Zhang, Xiangqi Kong, and Wangyang Wei. Concurrency optimized task scheduling for workflows in cloud. In *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, pages 709–716. IEEE, 2013.

[9] Nicolas G. Grounds. SOASim: Simulator for distributed system scheduling. http://soasim.sourceforge.net/, 2010–2018.

[10] Nicolas G. Grounds and John K. Antonio. A model-based scheduling framework for enhancing robustness. In *PDPTA*, 2018.

[11] Nicolas G. Grounds, John K. Antonio, and Jeffrey T. Muehring. Cost-minimizing scheduling of workflows on a cloud of memory managed multicore machines. In *CloudCom*, 2009.

[12] Khondker Shajadul Hasan, Nicolas G Grounds, and John K Antonio. Predicting cpu availability of a multi-core processor executing concurrent java threads. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2011.

[13] Matthew Hertz. *Quantifying and improving the performance of garbage collection*, volume 67. Citeseer, 2006.

[14] Matthew Hertz and Emery D Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *ACM SIGPLAN Notices*, volume 40, pages 313–326. ACM, 2005.

[15] Jong-Kook Kim, Sameer Shivle, Howard Jay Siegel, Anthony A Maciejewski, Tracy D Braun, Myron Schneider, Sonja Tideman, Ramakrishna Chitta, Raheleh B Dilmaghani, Rohit Joshi, et al. Dynamic mapping in a heterogeneous environment with tasks having priorities and multiple deadlines. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 15–pp. IEEE, 2003.

[16] Kun Li, Gaochao Xu, Guangyu Zhao, Yushuang Dong, and Dan Wang. Cloud task scheduling based on load balancing ant colony optimization. In *2011 Sixth Annual ChinaGrid Conference*, pages 3–9. IEEE, 2011.

[17] Sorin Manolache, Petru Eles, and Zebo Peng. Task mapping and priority assignment for soft real-time applications under deadline miss ratio constraints. *ACM Trans. Embed. Comput. Syst.*, 7(2):19:1–19:35, January 2008.

[18] Matthew Martin. Deadlock avoidance in distributed service oriented architectures. Master's thesis, University of Oklahoma, 2011.

[19] Matthew Martin, Nicolas G. Grounds, John K. Antonio, Kelly Crawford, and Jason Madden. Banker's deadlock avoidance algorithm for distributed service-oriented architectures. In *PDPTA*, 2010.

[20] Sung-Heun Oh and Seung-Min Yang. A modified least-laxity-first scheduling algorithm for real-time tasks. In *Real-Time Computing Systems and Applications, 1998. Proceedings. Fifth International Conference on*, pages 31–36. IEEE, 1998.

[21] Suraj Pandey, Linlin Wu, Siddeswara Mayura Guru, and Rajkumar Buyya. A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments. In *Advanced information networking and applications (AINA), 2010 24th IEEE international conference on*, pages 400–407. IEEE, 2010.

[22] Mohsen Amini Salehi, Jay Smith, Anthony A Maciejewski, Howard Jay Siegel, Edwin KP Chong, Jonathan Apodaca, Luis D Briceño, Timothy Renner, Vladimir Shestak, Joshua Ladd, et al. Stochastic-based robust dynamic resource allocation for independent tasks in a heterogeneous computing system. *Journal of Parallel and Distributed Computing*, 97:96–111, 2016.

[23] Vahid Salmani, Mahmoud Naghibzadeh, Amirali Habibi, and Hossein Deldari. Quantitative comparison of job-level dynamic scheduling policies in parallel real-time systems. In *TENCON 2006. 2006 IEEE Region 10 Conference*, pages 1–4. IEEE, 2006.

[24] Hira Shrestha, Nicolas G. Grounds, Jason Madden, Matthew Martin, John K. Antonio, Jay Sachs, Josh Zuech, and Carlos Sanchez. Scheduling workflows on a cluster of memory managed multicore machines. In *PDPTA*, 2009.

[25] Brinkley Sprunt, Lui Sha, and John Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.

[26] Amandeep Verma and Sakshi Kaushal. Deadline and budget distribution based cost-time optimization workflow scheduling algorithm for cloud. In *IJCA Proceedings on international conference on recent advances and future trends in information technology (iRAFIT 2012)*, volume 4, pages 1–4. iRAFIT (7), 2012.

[27] Qishi Wu, Daqing Yun, Xiangyu Lin, Yi Gu, Wuyin Lin, and Yangang Liu. On workflow scheduling for end-to-end performance optimization in distributed network environments. In Walfredo Cirne, Narayan Desai, Eitan Frachtenberg, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 76–95, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[28] Lingfang Zeng, Bharadwaj Veeravalli, and Xiaorong Li. Scalestar: Budget conscious scheduling precedence-constrained many-task workflow applications in cloud. In *Advanced Information Networking and Applications*

(AINA), 2012 IEEE 26th International Conference on, pages 534–541. IEEE, 2012.

[29] Yuanyuan Zhang, Wei Sun, and Yasushi Inoguchi. Predicting running time of grid tasks based on cpu load predictions. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, pages 286–292. IEEE Computer Society, 2006.

[30] Chenhong Zhao, Shanshan Zhang, Qingfeng Liu, Jian Xie, and Jicheng Hu. Independent tasks scheduling based on genetic algorithm in cloud computing. In *Wireless Communications, Networking and Mobile Computing, 2009. WiCom'09. 5th International Conference on*, pages 1–4. IEEE, 2009.

DEDICATION

to

My wife

Natalie Grounds

For

Sticking with me through these nine years of post-graduate studies

and
My God and His son, Jesus

For

Granting me the capability and opportunities necessary to achieve this and all other things