Bid Based Scheduler with Backfilling for a Multiprocessor System

Inbal Yahav
University of Maryland,
College Park, MD
iyahav@rhsmith.umd.edu

Louiza Raschid
University of Maryland,
College Park, MD
louiza@umiacs.umd.edu

Henrique Andrade
IBM Research,
Hawthorne, NY
hcma@us.ibm.com

ABSTRACT

We consider a virtual computing environment that provides computational resources on demand to users with multi-attribute task descriptions that include a valuation, resource (CPU) needs and a completion deadline. Achieving a high quality of service in this environment depends on finding a balance between processing high priority tasks before their deadlines expire, while maximizing resource utilization. The problem becomes more challenging in an economic setting, where the task valuation and deadline is private. We propose a bid-based server that publishes a history of the success rate table (SRT) for processed tasks. Clients use the history to optimize their bid for resources on a (single) multiprocessor server. The server schedules tasks in descending order of their bid and backfills the schedule with smaller tasks when resources are still available. The scheduler follows a hard deadline model where tasks cannot be processed after their deadline. We study three variations of the SRT where bidding history is published at different granularities. Using a simulation based study, we analyze the behavior of clients’ bids for the three SRT variants. We use the bid behavior to understand server performance. Our results show that publishing the SRT at a coarse granularity can lead to better weighted throughput and profit and preserve client fairness (with regards to clients’ CPU needs). The server publishing an SRT at a finer granularity shows better resource utilization.

1. INTRODUCTION

Users and organizations often face the issue of servicing workloads that exceed their local computational capabilities. Typically, seasonal trends as well as spikes in demand are challenging, if organizations rely solely on resources that are locally available. In many cases, obtaining additional computational resources rely on outsourcing, by leveraging external providers, is cost effective. This need for on-demand access to computing gave rise to the computational Grid paradigm [6]. In essence, a computational Grid enables users to draw resources transparently from multiple distributed providers.

Matching the diversity of user types and needs (e.g., differences in job priorities, completion deadlines, resource requirements, and processing time) to servers’ capability limitations results in a non-trivial parallel, real-time scheduling problem faced by the service providers. The online scheduling of multi-attribute jobs has some interesting implementations. For example, consider the Amazon Elastic Compute Cloud infrastructure (Amazon EC2) [1] that offers a virtual computing environment to users in return for a flat fee. In that setting, attributes such as the job deadline and resource needs as well as whether the job was effectively processed within the established constraints are variables known only to the user. This is limiting in at least two ways. First, competing resource providers cannot dynamically set their pricing strategies as a function of market conditions such as resource scarcity or urgency. Along the same line, users cannot shop around as market conditions, including pricing trends, are not publicly available. Under these circumstances, commonly used scheduling policies are first-in-first-out (FIFO) and shortest-job-first (SJF). Both provide a degree of fairness, yet fail to employ additional job properties and, hence, cannot be used to enable a more sophisticated market.

Many parallel scheduling policies have been developed and studied with the goal of maximizing resource utilization. Traditional scheduling policies for a single multiprocessor server commonly focus on queueing policies, dealing with issues such as jobs preemption [4, 9], where job processing can be stopped and restarted and global queueing versus multiple queues [5, 8, 15]. For example, Feitelson and Rudolph [5] suggest automatic load sharing as an advantage of global queue. Lawson and Smirni [8] introduce the duration based job classification into multiple queues to reduce the average slowdown experienced by jobs. More recent research focuses on backfilling policies, which combine a reservation for a given job in the interest of progress, with packing of additional jobs to fill in the allocation holes and, therefore, increase overall utilization [8, 12, 14, 16]. Backfilling is currently used by many schedulers, including IBM’s LoadLeveler [10], the Maui scheduler [11] and the PBS scheduler [13].

In this paper we consider a parallel scheduling mechanism in conjunction with the publication of bidding information associated with jobs that have previously been submitted for execution. We study the problem of welfare maximization, in which the goal is to maximize the total value of processed tasks under the constraints of resource capabilities and job...
completion deadlines.

We propose a policy referred to as Highest-Bid-First (HBF); it is a history-based policy where the server publishes recent processing success rate as a function of customer bids. Such information can be used by future users who expect to submit jobs to the system. We consider three variations to the HBF policy, HBF2Dim, HBF2Dim_perUnit, and HBF3Dim. These HBF variants differ by the level of granularity of the success rate (SRT) that the server makes public. HBF also uses backfilling. The main contributions of this paper are as follows:

- A description and analysis of the HBF scheduling policy with backfilling. The policy is geared towards optimizing the server’s weighted throughput. We show that it yields better performance than HBF without backfilling; the latter has been shown to be the best performer when no public information is made available for a simpler setting [17].
- The simulation-based analysis of how public bidding information affects future bidding behavior, and hence, the server’s economic outcome as measured by the observed weighted throughput and profit.

The rest of the paper is organized as follows. In Section 2, we provide a formal definition of the online parallel scheduling problem in an economic setting. In Section 3, we study the off-line variation of the throughput and weighted throughput maximization problem. In Section 4, we introduce an online bid-based with backfilling scheduling method for allocating parallel resources. We also discuss number of variations for the bidding policy. We study the empirical performance of the algorithms in Section 5. Finally, in Section 6, we conclude and suggest extensions to the techniques outlined in this paper.

2. MODEL AND PROBLEM DEFINITION

Typically, a virtual computing environment includes a network of servers providing computing resources and data storage that has to be shared among a variety of users. Users in these environments commonly submit tasks to the servers, characterized, for example, by valuation, completion deadline, and resource needs (e.g., CPU, storage).

We consider a simplified model in which the environment is composed of a single multiprocessor server with $n$ CPUs. Tasks\(^1\) in our setting have multi-attributes that include a valuation, resource (CPU) needs and a completion deadline. A server needs to schedule tasks to be served at each time slot. For simplicity, we assume an identical expected processing time of 1 time slot for each task.

We use $T = \{t_i\}$ to denote the set of tasks, with $t_i$ being the $i^{th}$ arriving task. We identify each task by the attributes \{$r_i, d_i, v_i, c_i$\}, representing respectively:

- $r_i$: Release time
- $d_i$: Deadline, after which the task is no longer executed (otherwise known as hard deadline model)
- $v_i$: Value

\(^1\)We employ the terms job and task interchangeably throughout this paper.

- $c_i$: Number of required CPU ($c_i \in [1, n]$)

We define a span $s_i$ of a task to be the difference between the deadline and the release time, i.e. the time to wait in the queue before abandoning it. We compute a bid $b_i$ to be described in Section 4. We use $Q(T) = t^k$ to represent the ordered set of tasks in the queue, with $t^k = \{t^k, b^k, c^k\}$ being the $k^{th}$ task in the queue, and \{$t^k, b^k, c^k$\} the task information that is shared with the server (value and deadline are considered as private information).

Time is divided into discrete time slots, represented by $N = \{1, 2, 3, ...\}$.

We define $x_{ij}$ to be a Boolean variable to denote whether $t_i$ was processed at time $j$. Note that each task can be processed at 1 time slot only ($\forall i, \sum_{j \in N} x_{ij} \leq 1$). We use $v_{ij}$ and $c_{ij}$ to represent the value gained and the required CPU respectively from processing request $t_i$ at time $j$, where:

$$v_{ij} = \begin{cases} v_i & \text{if } r_i \leq j \leq d_i \\ 0 & \text{otherwise} \end{cases}$$

$$c_{ij} = \begin{cases} c_i & \text{if } r_i \leq j \leq d_i \\ 0 & \text{otherwise} \end{cases}$$

We consider the problem of a single multiprocessor weighted throughput (SMWT), where the goal is to maximize the average processed value per time slot of the multiprocessor server. The formalization of the problem is given by Eq. 1.

$$WTh = \max \frac{1}{|N|} \sum_{i \in T, j \in N} v_{ij}x_{ij}$$

We use the following metrics to evaluate the scheduling mechanism:

**Completion Rate (CR)** The average fraction of processed tasks:

$$CR = \frac{1}{|T|} \sum_{i \in T, j \in N} x_{ij}$$

**Utilization (U)** The average used fraction of CPU per time slot:

$$U = \frac{1}{|N|} \sum_{i \in T, j \in N} c_{ij}x_{ij}$$

We consider the scheduling problem in an online setting, implying that the server has no prior knowledge of future tasks. Table 1 summarizes the notation used in this paper.

3. OFFLINE OPTIMAL SCHEDULING

To better understand the scheduling problem, we first consider it in an offline setting in which there is complete a-priori information of all tasks and task attributes, e.g., value, CPU need and completion deadline, are publicly shared between the users and the server.

We use a bipartite graph representation for the problem as follows (see Figure 1 for an example):

- A set of nodes representing the tasks.
- A set of nodes representing the discrete processing time slots \{1,2, ...\}.
Arriving tasks

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Tasks set</td>
</tr>
<tr>
<td>tᵢ</td>
<td>The iᵗʰ arriving task</td>
</tr>
<tr>
<td>rᵢ</td>
<td>Release time for the iᵗʰ task</td>
</tr>
<tr>
<td>vᵢ</td>
<td>Value of the iᵗʰ task</td>
</tr>
<tr>
<td>cᵢ</td>
<td>CPU's requirement the iᵗʰ task</td>
</tr>
<tr>
<td>bᵢ</td>
<td>Bid of the iᵗʰ task</td>
</tr>
</tbody>
</table>

Queue

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>The kᵗʰ task in the queue</td>
</tr>
</tbody>
</table>

Proof. The SMWT problem is an offline setting NP-Complete [3].

The Knapsack problem is the following combinatorial optimization task: given a set of items, each with a cost and a value, determine the number of each item to include in the collection so that the total cost is less than some given cost and the total value is as large as possible. Formally [3]: There is a knapsack of capacity c > 0 and M items. Each item has value vᵢ > 0 and weight wᵢ > 0. Find the selection of items (δᵢ = 1 if selected, 0 if not) that fit, \( \sum_{i=1}^{M} \delta_i w_i \leq c \), and the total value, \( \sum_{i=1}^{M} \delta_i v_i \) is maximized.

Consider an offline scheduling with 1 time slot and the set of M tasks T that can be processed in that time slot: \( \forall t_i \in T : d_i \leq u \leq r_i \). We use u to represent the knapsack with capacity of 100%. The set of tasks in the offline scheduling problem is equivalent to the set of items in the Knapsack, where the iᵗʰ item value \( v_i \) is the value of the task and \( w_i \) is the CPU requirement. Maximizing the weighted throughput under CPU constraint in the offline scheduling problem, is equal to maximizing the total value under the capacity constraint in the Knapsack problem. \( \square \)

Unfortunately, the input to the integer programming problem can easily be very large (approximately: \( |N| + |T| \) nodes, \( |T| + |E(\text{span})| \) edges). Since the solution is exponential as a function of the input parameters, we cannot optimally solve large instances of the scheduling problem nor can we provide bounds except for trivially small configurations.

4. ONLINE SCHEDULING

We consider the SMWT problem in an online setting where the server has no knowledge of future arrivals. We first present a bid-based scheduling policy for the server, named Highest-Bid-First (HBF). This policy allows a client to prioritize their tasks in the queue by submitting the task and a bid.

To establish a baseline, we also study the performance of the well known greedy Earliest-Deadline-First (EDF) scheduling policy. EDF implicitly assumes public knowledge of task deadline and value.

4.1 Highest-Bid-First Scheduling Policy (HBF)

We consider online scheduling in an economic setting, where task attributes are privately known only to the owner. We assume that users are self-interested and, therefore, reveal only information that can benefit them.

We propose a bid-based policy that prioritizes tasks in a decreasing order of their bids (Highest-Bid-First) as follows:

- At each time slot the server processes the first task in the queue and, if capacity allows, backfills with smaller tasks from the queue, to fill allocation holes and maximize resource utilization.
- The server published a history of its task processing success, labeled success rate table (SRT). This is discussed next.
- A user computes a bid by maximizing its expected utility, given by \( E(u_i) = \max_4 (v_i - b) \times p \), where \( p \) is the probability of success, i.e., the probability that a task with a bid \( b \) completed within its span. This probability will be determined from the SRT.
- When a tasks is released, the owner submits the task to the server. The information provided to the server for

![Figure 1: Offline graph representation](image-url)
each task \( t' = \{ r', b', c' \} \), where \( r' \) is the release time, \( b' \) is the bid and \( c' \) is the CPU need. Note that the value and deadline for the task are private and not shared with the server.

- We use \( Q(T) = t^k \) to represent the ordered set of tasks in the server queue, with \( t^k = \{ r^k, b^k, c^k \} \) being the \( k \)th task in the queue.

In order for the bidding mechanism to achieve high weighted throughput and utilization, the following properties are desirable:

- Positive correlation between bids and values:
  \[ \forall v_i < v_j, \text{ if } (s_i = s_j \land c_i = c_j) \text{ then: } (b_i \leq b_j) \] (5)

- Negative correlation between bids and spans:
  \[ \forall s_i < s_j, \text{ if } (v_i = v_j \land c_i = c_j) \text{ then: } (b_i \geq b_j) \] (6)

- A sufficient spread of bids in value and spans that will allow the server to differentiate between different types of customers.

The server has the incentive to assist the clients to present a bid that will maximize the probability that the tasks will be successfully completed prior to the deadline. On the other hand, maintaining a history of the server’s success rate (SRT) is an overhead. We study three variations of the SRT information that is shared by the server. They are HBF2Dim, HBF2Dim_perUnit and HBF3Dim. They vary by the granularity of the SRT information.

**HBF2Dim.** The server publishes a 2-dimensional SRT, corresponding to bids and spans\(^2\). This level of granularity allows clients to bid as a function of span, assuming that customers with tasks of higher value have the resources to bid higher. Figure 2 depicts a sample table where each cell represents the success rate for combinations of bid and span. The expression to compute the bid is as follows: \( \max_i (v_i - b) \ast p(\text{getting processed} | b, s) \)

**HBF2Dim_perUnit.** The server publishes the same structure information to as in HBF2Dim at the granularity of bid per CPU unit as seen in Figure 3. Consequently, the bid placed by the user will be directly proportional to the number of CPU units required and is as follows: \( \max_i (v_i - c_i \ast b) \ast p(\text{getting processed} | b, s) \)

\(^2\)Note that calculating the SRT using span data does not conflict with the private knowledge assumption since the information is published retroactively. Alternatively, the SRT can be published by a third party.

**HBF3Dim.** HBF3Dim publishes a 3-dimensional SRT and provides the success rate for each value of bid, span, and CPU requirement as seen in Figure 4. HBF3Dim provides precise and complete information about successful bids to clients, allowing them to finely tune their bids based additionally on CPU needs. The expression to compute the bid is as follows: \( \max_i (v_i - b) \ast p(\text{getting processed} | b, s, c) \)

To summarize, for HBF2Dim the bid function is a fraction of span and is independent of CPU, for HBF2Dim_perUnit the bid history is a function of span and is directly proportional to CPU, and for HBF3Dim the bid history is a function of both span and CPU.

**4.2 Early-Deadline-First (EDF)**

For a baseline, we examine the traditional Early-Deadline-First (EDF) policy. Our previous results on task scheduling show that under hard deadline model and a single processor server, the simple greedy algorithm EDF is optimal in terms of the task completion rate [17]. Yet, the results do not apply on a multiprocessor server, where tasks differ in their CPU requirement. For example, consider the following task arrival scenario:

- Task \( t_1 \): \( d_1 = 1, c_1 = 0.6n \)
- Task \( t_2 \): \( d_2 = 2, c_2 = 0.8n \)
- Task \( t_3 \): \( d_3 = 3, c_3 = 0.4n \)
- Task \( t_4 \): \( d_4 = 3, c_4 = n \)

A traditional EDF scheduler will process tasks \( t_1 \) and \( t_2 \) at time slot 1 and 2 respectively and either task \( t_3 \) or \( t_4 \) at time slot 3. Clearly, executing tasks \( t_1 \) and \( t_2 \) together yields the optimal solution. To overcome this problem, we allow the scheduler to backfill tasks whenever there is an unused CPU. In the previous example, task \( t_3 \) can be processed during the first unit, along with task \( t_1 \).

Whereas EDF may have high utilization and completion rate performance, it is known to provide poor results in term of weighted throughput. In fact it is known to converge to...
4.3 Backfilling

We complement HBF and EDF scheduling policy with a backfilling method. The server backfills with tasks from the head of the queue, as long as the tasks do not exceed the server’s capacity. For example, consider the following queue:

Task t1: b1 = 10, c1 = 0.5n
Task t2: b2 = 9, c2 = 0.6n
Task t3: b3 = 8, c3 = 0.4n
Task t4: b4 = 7, c4 = 0.3n
Task t5: b5 = 6, c5 = 0.2n

The server will first accept task t1 and then it will backfill with task t5. The backfilling method is outlined in Figure 5. There are several variations for backfilling. For example, the server can choose the task with the highest bid on the queue that does not exceed its CPU capacity; this will improve throughput. Alternately, the server can choose the largest job on the queue whose CPU need does not exceed its capacity; this will improve server utilization. We experimented with both techniques but did not find significant differences in the results so we report on the first variation.

5. EXPERIMENTS AND RESULTS

In this section we discuss our experimental results. Section 5.1 describes our simulation setup environment. We then study the bidding behavior of the three variants of HBF policies (Section 5.3). Finally, in Section 5.4, we compare their performance using EDF as the baseline policy.

5.1 The Simulation Configuration

We implemented our simulation platform using Visual Studio.Net 2003 platform. Two main entities were simulated:

Tasks: Each task in our mechanism has a private valuation, completion time, and a CPU need. We assume an identical independent uniform distribution for values and spans. We experimented with different models for iid CPU requirement (Uniform workload, high workload, etc.) and found that the results were unchanged. We assume an identical processing time (1 unit) for all tasks. We model the arrival rate of tasks as a Poisson process with an intensity parameter \( \lambda \). The expected bid is rounded to a discrete value in the range [0, 10]. Table 2 summarizes the task setting in our experiments.

Server: The server publishes a statistics table of process success rates at its queue, named SRT, allowing the customers to calculate the best utility, based on history. The SRT is refreshed every \( y \) time slots. The calculation of the success rate for window \([t_x, t_{x+y}]\) is given by:

\[
S[x, x+y][b][s][< c >] = \text{The number of tasks that arrived in time window } [t_x, t_{x+y}], \text{ with a span of } s, \text{ a bid of } b \text{ and a CPU need of } c \text{ (for HBF3Dim only)}, \text{ that were successfully processed.}
\]

\[
T[x, x+y][b][s][< c >] = \text{The total number of requests that arrived in time window } [t_x, t_{x+y}], \text{ with a span of } s, \text{ a bid of } b \text{ and a CPU need of } c.
\]

In each re-computation, the new probability table uses a smoothing function combining new and old success rates, as followed:

\[
P_{t_{x+y}} \text{(getting processed } | b, s, c) = \alpha \cdot \frac{S[x, x+y][b][s][< c >]}{T[x, x+y][b][s][< c >]} + (1 - \alpha) \cdot P_{t_x} \text{(getting processed } | b, s, c)
\]

Parameters: A service rate of a server is set to be \( k = 1 \), meaning each task is processed completely in one time slot, or not processed at all. We set the number of CPUs to 20. Every \( y = 1000 \) time slots the server refreshes its SRT. We use \( \alpha = 0.9 \) for the smoothing function. The smoothing function by its nature gives higher priority to recent history.

Metrics: We use three metrics to evaluate performance. They are weighted throughput, completion rate, and utilization and the expressions are in Eq. 1, 2 and 3, respectively.

5.2 Results

We examine the bidding behavior of the tasks in response to the SRT variants revealed by the server. We present the bid distribution of tasks as a function of CPU need, span and value. We then compare the performance metrics of these variants to understand the impacts of the SRT on server performance.

5.3 Bidding Mechanism Properties

Consider the HBF2Dim server that reveals the SRT as a function of value and span. Figure 6 presents the bid distribution as a function of value for different span. Figure 7 presents the bid distribution as a function of span for different value(s). The figures show a positive correlation of bids with respect to both value and span. From Figure 6 we see that value has a strong differentiating impact on the

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Distribution</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_i )</td>
<td>Uniform</td>
<td>[1, 2 ... 10]</td>
</tr>
<tr>
<td>( s_i )</td>
<td>Uniform</td>
<td>[0, 1 ... 9]</td>
</tr>
<tr>
<td>( c_i )</td>
<td>-</td>
<td>[5%, 10% ... 100%]</td>
</tr>
<tr>
<td>( b_i )</td>
<td>-</td>
<td>[0, 10] ( \leq v_i )</td>
</tr>
</tbody>
</table>
bid. For example, for span= 0, the bid for high value tasks increases steeply. From value 7 to value 9 the bids goes up by 0.5 units. Similarly, as the value decreases, the bid falls rapidly. The impact of span to differentiate the bid value is marginal. There are roughly two groups of bid values and the the bids reduce slightly as span increases.

Next, we examine the bid distribution of the HBF2Dim policy, where the SRT is a function of bids, deadlines and CPU. The bid distribution by value is in Figure 8 and the distribution by span is in Figure 9. The figures show CPU need of 10% and 60%. Figure 8 shows a strong correlation between bids and values and Figure 9 shows a strong correlation with span. Both figures further show a strong correlation between bids and CPU need; for the same value of span and value, the bid for 60% CPU need is much higher than the bid for 10%.

We further note that bids are strongly differentiated by span for HBF3Dim, compared to HBF2Dim. For example, in Figure 9, for (CPU= 10%), the bid declines rapidly as span increases. The bids for HBF3Dim are also differentiated by value, but the the impact is less than HBF2Dim. If we compare Figure 6 and 8, the bids increase more rapidly with increasing value in HBF2Dim. The bid distribution for HBF2Dim_perUnit, which requires the customers to bid proportionately to CPU need, is given in Figures 10 and 11. We can see that bids for many tasks under this policy, for high CPU need (60%), are zero. These tasks have low incentive to bid due to low probability of getting processed. This behavior is independent of value and span.

5.4 Performance

Based on our observations of bid behavior, we expect the queues for the three SRT variants to be characterized as follows:

**HBF2Dim** Very high value tasks will dominate the head of the queue while low tasks will congregate at the tail, since value is a strong determinant. Within each group, low span tasks will take priority over high span. We expect that CPU need will be uniformly distributed among the tasks in the queue since this policy is transparent to CPU needs.

**HBF2Dim_perUnit** Recall that for this policy, bids are in direct proportion to CPU need as well as value and span. Consequently, the queue will be dominated by medium and low CPU need tasks and many high CPU need tasks will not be able to make a feasible bid. Since neither value nor span can differentiate bids very strongly for this policy, we expect high value and low span tasks to dominate at the head of the queue, while all other tasks will not be well differentiated. High CPU tasks will tend to be found at the end of the queue.

**HBF3Dim** Recall that value, span and CPU need have an impact for this policy. The head of the queue will contain high CPU, high value and low span tasks. The tail will contain low CPU, low value and high span tasks. The other tasks will be spread through the queue.

We expect HBF2Dim and HBF3Dim to have high weighted throughput since value had a strong impact on both policies. The former should perform better since value differentiated the bids most significantly for HBF2Dim. We expect HBF3Dim and HBF2Dim_perUnit to have higher utilization since CPU need could impact the bid, whereas, bids were not correlated with HBF2Dim. The best policy for task completion rate is expected to be HBF2Dim_perUnit since it is dominated by low and medium CPU need tasks and few high CPU need tasks are admitted. Thus, it processes more tasks. Finally, since HBF2Dim has two desirable properties - value strongly differentiates bids, and CPU needs are evenly distributed in the queue, HBF2Dim shows fairness towards tasks’ CPU needs and simultaneously, will maximize profit for the server. Profit corresponds to the sum of the bids.

5.5 Performance Comparison

After analyzing the individual performance of the three policies we compare their performance and EDF performance in terms of completion rate, utilization and weighted throughput for different arrival rates. Note that arrival rate \( \approx 2.8 \) gives 100% load \( \approx 100\% \approx 100\% \). From now on we refer to arrival rates that are much smaller then 2.8 as low load, arrival rate around 2.8 as normal load and the others as heavy load.

First, we present in Figure 12 the completion rate of the four methods as a function of arrival rate. As expected,
**HBF3Dim**

**CPU requirement = 10%**

![Graph](image1)

**CPU requirement = 60%**

![Graph](image2)

Figure 8: Average bid per value

**HBF3Dim**

**CPU requirement = 10%**

![Graph](image3)

**CPU requirement = 60%**

![Graph](image4)

Figure 9: Average bid per value

**HBF2Dim_perUnit**

**CPU requirement = 10%**

![Graph](image5)

**CPU requirement = 60%**

![Graph](image6)

Figure 10: Average bid per value
among the HBF various methods HBF2Dim_perUnit performs better for heavy load. For normal and low loads, where deadline has more importance than grouping tasks together, HBF2Dim_perUnit performs less good. Interestingly, for heavy loads HBF2Dim_perUnit compares to EDF. The former tends to sort tasks by value and CPU need, whereas the later cares about spans only.

Next, we compare the utilization of the methods in Figure 13. Not surprisingly, for low and normal loads, EDF outperforms all other methods. For heavy load, where the queue is always full, all methods show equal performance. Among the HBF methods, HBF3Dim gives the best performance, since it better differentiate spans of tasks.

We compare the methods in terms of weighted throughput in Figure 14. The three HBF variants clearly outperform EDF. HBF2Dim and HBF3Dim exhibit the best performance, as expected.

We also compare HBF2Dim with and without backfilling in Figure 15. Our previous results [17] show that HBF
performs as well as the best-known scheduling algorithm (RMIX [2]) with regard to weighted throughput, for the simpler setting where all tasks have identical CPU requirements. Note that HBF keeps the deadline private while RMIX assumes that the deadline is public. In Figure 15, we present the weighted throughput normalized by the completion rate for each policy. The X axis is the arrival rate. We normalize both (with respect to the completion rate) to make it a fair comparison since the mix of CPU needs processed by HBF2Dim varies. The results show that their behavior is almost indistinguishable, and indicates that HBF2Dim with backfilling maintains good efficiency of processing tasks.

Finally, we compare the server profit per unit in Figure 16. The figure show that HBF2Dim gains the highest profit, followed by HBF3Dim.

To summarize the performance observations, when the server provides history at a lower level of granularity (HBF3Dim), higher bids are placed for tasks with high CPU need, high value and early deadline. As a result, there is more competition of tasks with high CPU need against tasks with high value and early deadline (for HBF3Dim). Since bids in HBF2Dim are independent of CPU need, high CPU tasks are not favored. This leads to greater fairness with respect to processing CPU needs, better overall weighted throughput, and better profit for HBF2Dim. It appears best suited to trade-off the need to service high bid jobs before their deadline, while being fair to all tasks. HBF2Dim perUnit does not perform well. This policy forces bids that are directly proportional to CPU need. As a result, bids for high CPU tasks often exceed their value.

6. CONCLUSIONS

The paper introduced three variations of a Highest-Bid-First scheduling policy, a history-based policy, in which the server publishes recent processing success rate as a function of customer bids to aid future customers in tuning their bidding behavior as they submit new jobs. We also proposed a backfilling heuristic to support differences in customer CPU requirements. We studied the policies in terms of completion rate, utilization, and weighted throughput. We studied the relationship between the above measurements and the behavior of the customer bids. We showed that monotonicity of bids in values and spans results in high weighted throughput. Utilization, on the other hand, is strongly dependent on monotonicity in spans only.

Our results include comparison between the three variations, and an EDF-based server as well as a regular HBF-based server (i.e., no backfilling). With respect to weighted throughput, we showed that HBF2Dim is the best among the three variations. Moreover, it is comparable in terms of performance to the HBF policy.

We plan to expand this work in two manners. First, we will consider multiple queues instead of a single job submission queue. Second, we plan to add the estimated duration as an additional attribute to the task description. We in-
tend to model and evaluate different bidding mechanisms for different tasks durations, redesigning the backfilling policies appropriately. Additionally, we plan to extend a real scheduling system such as the Maui scheduler and experiment with real-world workloads.

7. REFERENCES