Core Algorithms of the Maui Scheduler

David Jackson, Quinn Snell, and Mark Clement

Brigham Young University, Provo, Utah 84602
jacksond@supercluster.org, {snell, clement}@cs.byu.edu

Abstract. The Maui scheduler has received wide acceptance in the HPC community as a highly configurable and effective batch scheduler. It is currently in use on hundreds of SP, O2K, and Linux cluster systems throughout the world including a high percentage of the largest and most cutting edge research sites. While the algorithms used within Maui have proven themselves effective, nothing has been published to date documenting these algorithms nor the configurable aspects they support. This paper focuses on three areas of Maui scheduling, specifically, backfill, job prioritization, and fairshare. It briefly discusses the goals of each component, the issues and corresponding design decisions, and the algorithms enabling the Maui policies. It also covers the configurable aspects of each algorithm and the impact of various parameter selections.

1 Introduction

The Maui scheduler [1] has received wide acceptance in the HPC community as an highly configurable and effective batch scheduler. It is currently in use on hundreds of IBM SP-2, SGI Origin 2000, and Linux cluster systems throughout the world including a high percentage of the largest and most cutting edge research sites. While Maui was initially known for its advance reservation and backfill scheduling capabilities, it also possesses many additional optimizations and job management features. There are many aspects of the scheduling decision which must be addressed. This paper documents the underlying algorithms associated with the Maui scheduler. While Maui originated as a project designed to purely maximize system utilization, it rapidly evolved into a tool with a goal of maximizing scheduling performance while supporting an extensive array of policy tools. The words *performance* and *policy* go a long way to complicating this problem.

2 Overview

Maui, like other batch schedulers [2,3,4], determines when and where submitted jobs should be run. Jobs are selected and started in such a way as to not only enforce a site's mission goals, but also to intelligently improve resource usage and minimize average job turnaround time. Mission goals are expressed via a combination of policies which constrain how jobs will be started. A number of base concepts require review to set the groundwork for a detailed discussion of the algorithms.

D.G. Feitelson and L. Rudolph (Eds.): JSSPP 2001, LNCS 2221, pp. 87–102, 2001.

2.1 Scheduling Iteration

Like most schedulers, Maui schedules on a iterative basis, scheduling, followed by a period of sleeping or processing external commands. Maui will start a new iteration when one or more of the following conditions is met:

- a job or resource state-change (i.e. job termination, node failure) event occurs
- a reservation boundary event occurs
- the scheduler is instructed to resume scheduling via an external command
- a configurable timer expires

2.2 Job Class

Maui supports the concept of a job class, also known as a job queue. Each class may have an associated set of constraints determining what types of jobs can be submitted to it. These constraints can limit the size or length of the job and can be associated with certain default job attributes, such as memory required per job. Constraints can also be set on a *per-class* basis specifying which users, groups, etc., can submit to the class. Further, each class can optionally be set up to only be allowed access to a particular subset of nodes. Within Maui, all jobs are associated with a class. If no class is specified, a default class is assigned to the job.

2.3 QoS

Maui also supports the concept of quality of service (QoS) levels. These QoS levels may be configured to allow many types of special privileges including adjusted job priorities, improved queue time and expansion factor targets, access to additional resources, or exemptions from certain policies. Each QoS level is assigned an access control list (ACL) to determine which users, groups, accounts, or job classes may access the associated privileges. In cases where a job may possess access to multiple QoS levels, the user submitting the job may specify the desired QoS. All jobs within Maui are associated with a QoS level. If no QoS is specified, a default QoS is assigned.

2.4 Job Credentials

Each batch job submitted to Maui is associated with a number of key attributes or credentials describing job ownership. These credentials include the standard user and group ID of the submitting user. However, they also include an optional account, or project, ID for use in conjunction with allocation management systems. Additionally, as mentioned above, each job is also associated with a job class and QoS credential.

2.5 Throttling Policies

Maui's scheduling behavior can be constrained by way of throttling policies, policies which limit the total quantity of resources available to a given credential at any given moment. The resources constrained include things such as processors, jobs, nodes, and memory. For example, a site may choose to set a throttling policy limiting the maximum number of jobs running simultaneously per user to 3 and set another policy limiting the group, staff, to only using a total of 32 processors at a time. Maui allows both *hard* and *soft* throttling policy limits to be set. Soft limits are more constraining than hard limits. Each iteration, Maui attempts to schedule all possible jobs according to soft policy constraints. If idle resources remain, Maui will re-evaluate its queue and attempt to run jobs which meet the less constraining hard policies.

3 Scheduling Iterations

On each scheduling iteration, Maui obtains fresh resource manager information, updates its own state information, and schedules selected jobs. These activities are broken down into the following general steps:

1. Obtain updated resource manager information. Calls are issued to the resource manager to get up-to-date detailed information about node and job state, configuration, etc.
2. Update statistics. Historical statistics and usage information for running jobs are updated. Statistics records for completed jobs are also generated.
3. Refresh reservations. Maui adjusts existing reservations incorporating updated node availability information by adding and removing nodes as appropriate. Changes in node availability may also cause various reservations to slide forward or backward in time if the reservation timeframe is not locked down. Maui may also create or remove reservations in accordance with configured reservation time constraints during this phase. Finally, idle jobs which possess reservations providing immediate access to resources are started in this phase.
4. Select jobs meeting minimum scheduling criteria. A list is generated which contains all jobs which can be feasibly scheduled. Criteria such as job state, job holds, availability of configured resources, etc. are taken into account in generating this list. Each job's compliance with various throttling policies is also evaluated with violating jobs eliminated from the feasible job list.
5. Prioritize feasible jobs. The list of feasible jobs is prioritized according to various job attributes, scheduling performance targets, required resources, and historical usage information.
6. Schedule jobs in priority order. Jobs which meet *soft* throttling policy constraints are selected and then started sequentially in a highest-priority-first order. When the current highest priority idle job is unable to start due to a lack of resource availability, the existing reservation space is analyzed and

the earliest available time at which this job can run is determined. A reservation for this job is then created. Maui continues processing jobs in priority order, starting the jobs it can and creating reservations for those it can't until it has made reservations for the top N jobs where N is a site configurable parameter.

7. Soft policy backfill. With the priority FIFO phase complete, Maui determines the current available *backfill windows* and attempts to best fill these holes with the remaining jobs which pass all soft throttling policy constraints. The configured backfill algorithm and metric is applied when filling these windows.
8. Hard policy backfill. If resources remain after the previous backfill phase, Maui selects jobs which meet the less constraining *hard* throttling policies and again attempts to schedule this expanded set of jobs according to the configured backfill algorithm and metric.

4 Backfill

Backfill is a scheduling optimization which allows a scheduler to make better use of available resources by running jobs out of order. When Maui schedules, it prioritizes the jobs in the queue according to a number of factors and then orders the jobs into a highest-priority-first sorted list. It starts the jobs one by one stepping through the priority list until it reaches a job which it cannot start. Because all jobs and reservations possess a start time and a wallclock limit, Maui can determine the completion time of all jobs in the queue. Consequently, Maui can also determine the earliest the needed resources will become available for the highest priority job to start.

Backfill operates based on this earliest-job-start information. Because Maui knows the earliest the highest priority job can start, and which resources it will need at that time, it can also determine which jobs can be started without delaying this job. Enabling backfill allows the scheduler to start other, lower-priority jobs so long as they do not delay the highest priority job. If Backfill is enabled, Maui, protects the highest priority job's start time by creating a job reservation to reserve the needed resources at the appropriate time. Maui then can start any job which will not interfere with this reservation.

Backfill offers significant scheduler performance improvement. Both anecdotal evidence and simulation based results indicate that in a typical large system, enabling backfill will increase system utilization by around 20% and improve average job turnaround time by an even greater amount. Because of the way it works, essentially filling in holes in node space, backfill tends to favor smaller and shorter running jobs more than larger and longer running ones. It is common to see over 90% of these small and short jobs backfilled as is recorded in the one year CHPC workload trace [5]. Consequently, sites will see marked improvement in the level of service delivered to the small, short jobs and only moderate to no improvement for the larger, long ones.

Suspicions arise regarding the use of backfill. Common sense indicates that in all systems there must be a tradeoff. In scheduling systems this tradeoff gen-

erally involves trading system utilization for fairness, or system utilization for turnaround time. However, tradeoffs are not always required. While it is true that tradeoffs are generally mandatory in a highly efficient system, in a less efficient one, you can actually get something for nothing. Backfill takes advantage of inefficiencies in batch scheduling actually improving system utilization and job turnaround time and even improving some forms of fairness such balancing average expansion factor distribution along a job duration scale.

4.1 Backfill Drawbacks

While backfill scheduling is advantageous, minor drawbacks do exist. First, the ability of backfill scheduling to select jobs out of order tends to dilute the impact of the job prioritization algorithm in determining which jobs are most important. It does not eliminate this impact, but does noticeably decrease it.

Another problem, widely ignored in the HPC realm, is that in spite of reservations to protect a job's start time, backfill scheduling can actually delay a subset of backlogged jobs. The term delay is actually inaccurate. While the start time of a job with a reservation will never slide back in time, backfill can prevent it from sliding forward in time as much as it could have otherwise, resulting in a *psuedo-delay*. This behavior arises through the influence of inaccuracies in job run time estimates and resulting wallclock limits. When a user submits a job, he makes an estimate of how long the job will take to run. He then pads this estimate to make certain that the job will have adequate time to complete in spite of issues such as being assigned to slow compute resources, unexpectedly long data staging, or simply unexpectedly slow computation. Because of this padding, or because of poor initial estimates, wallclock limits have been historically poor, averaging approximately 20 to 40% across a wide spectrum of systems. Feitelson reported similar findings [6] and the online traces at supercluster.org for the Center for High Performance Computing at the University of Utah and the Maui High Performance Computing Center show wallclock accuracies of 29.4% and 33.5% respectively.

This problem is exhibited in a simple scenario shown in Figure 1 involving a six-node system with a running job on 4 nodes, job A, which estimates its completion time will be in 3 hours. Two jobs are then queued, job B, requiring five nodes, cannot start until job A completes while job C requires only two nodes and three hours of walltime. A standard backfill algorithm would reserve resources for job B and then start job C. Now, lets assume the wallclock estimate of job A is off and it actually completes one hour early. Job B still cannot run because job C is now using one of its needed nodes. Because backfill started job C out of order, the start of the higher priority job B was actually delayed from its *potential* start time by one hour.

This is not a significant problem and is outweighed by the positive effects of backfill. Studies have shown that across a number of systems, only a small percentage of jobs are truly delayed. Figure 2 is representative of these results. To obtain this information, a large job trace from the Maui High Performance Computing Center was run with and without backfill enabled. The differences

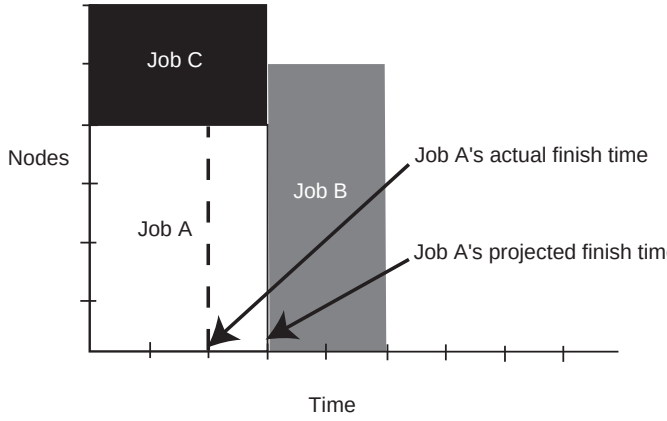

Fig. 1. Wallclock accuracy induced backfill delays.

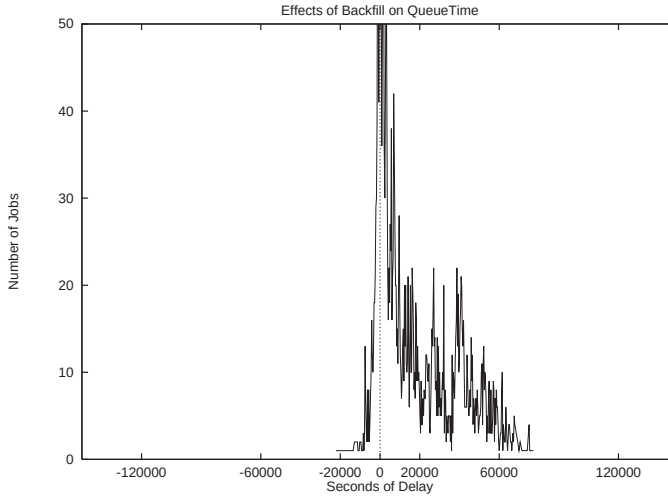

Fig. 2. Actual queue-time delay resulting from backfill based on inaccurate walltime estimates.

in individual queue times were calculated and plotted. Roughly 10% of the jobs experience a greater queue time with backfill enabled. These results are further examined in forthcoming studies. The percentage of delayed jobs is reduced by two primary factors. First, backfill results in general improvements in system utilization and job turnaround time for all jobs, not just those that are actually backfilled. This is because even jobs which are not backfilled are often blocked from running by other jobs which do get backfilled. When the blocking job is

started early, the blocked job also gets to start earlier. Its a classic case of *a rising tide lifts all ships* and virtually every job benefits. The second relevant factor is that wall clock limit inaccuracies are widespread. The 2D bin packing view of an HPC system where the start time of each job can be effectively calculated out to infinity is grossly misleading. The real world situation is far more sticky with jobs constantly completing at unexpected times resulting in a constant reshuffling of job reservations. Maui performs these reservation adjustments in a priority order allowing the highest priority jobs access to the newly available resources first, thus providing a mechanism to favor priority jobs with every early job completion encountered. This priority based evaluation consequently provides priority jobs the best chance of improving their start time. Thus, priority based reservation adjustment counters, as far as possible, the wallclock accuracy psuedo-delays.

Given the pros and cons, it appears clear for most sites that backfill is definitely worth it. Its drawbacks are rare and minor while its benefits are widespread and significant.

4.2 Backfill Algorithm

The algorithm behind Maui backfill scheduling is mostly straightforward although there are a number of issues and parameters to be aware of. First of all, Maui makes two backfill scheduling passes. For each pass, Maui selects a list of jobs which are eligible for backfill according to the user specified throttling policy limits described earlier. On the first pass, only those jobs which meet the constraints of the *soft* policies are considered and scheduled. The second pass expands this list of jobs to include those which meet the less constraining *hard* fairness throttling policies.

A second key concept regarding Maui backfill is the concept of backfill windows. Figure 3 shows a simple batch environment containing two running jobs and a reservation for a third job. The present time is represented by the leftmost end of the box with the future moving to the right. The light gray boxes represent currently idle nodes which are eligible for backfill. To determine backfill windows, Maui analyzes the idle nodes essentially looking for largest node-time rectangles. In the case represented by figure 2, it determines that there are two backfill windows. The first window contains only one node and has no time limit because this node is not blocked by any reservation. The second window, Window 2, consists of 3 nodes which are available for two hours because some of the nodes are blocked by a reservation. It is important to note that these backfill windows partially overlap yielding larger windows and thus increasing backfill scheduling opportunities.

Once the backfill windows have been determined, Maui begins to traverse them. By default, these windows are traversed widest window first but this can be configured to allow a longest window first approach to be employed. As each backfill window is evaluated, Maui applies the backfill algorithm specified by the BACKFILLPOLICY parameter, be it FIRSTFIT, BESTFIT, etc.

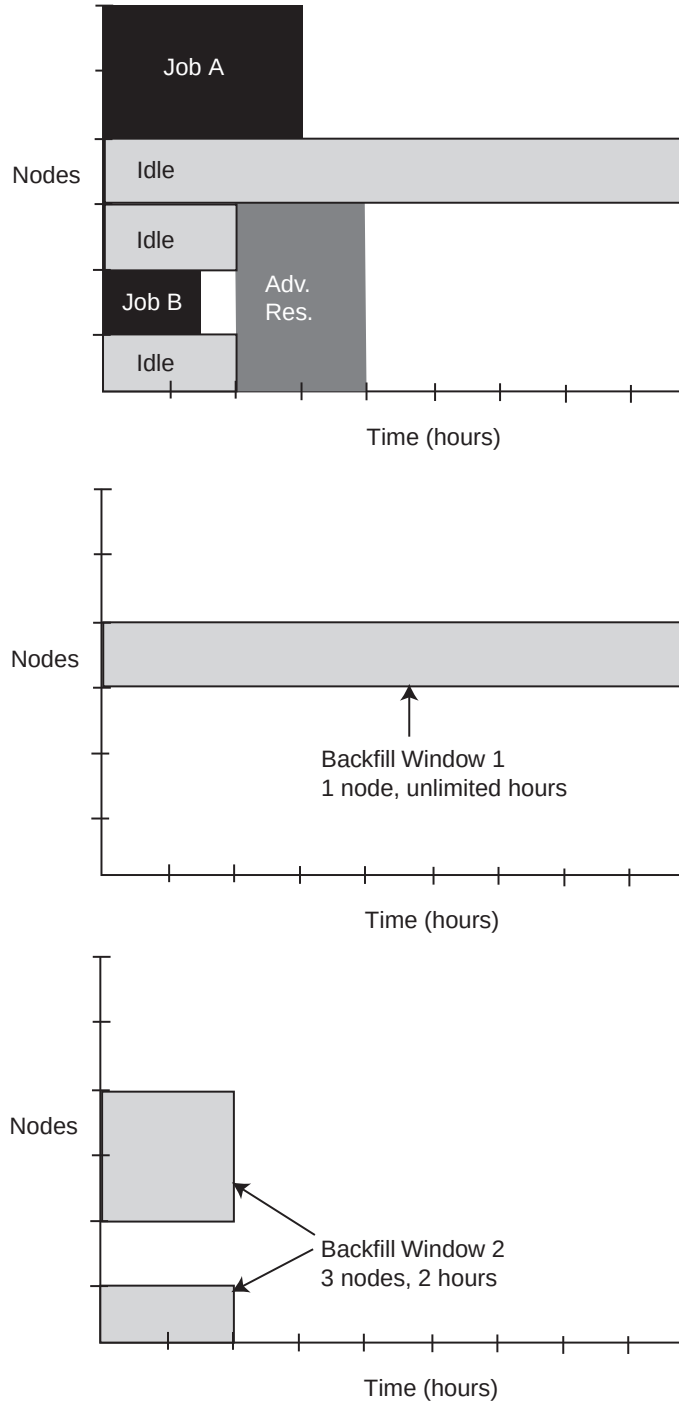

Fig. 3. Backfill Windows.

Assuming the BESTFIT algorithm is applied, the following steps are taken.

1. The list of feasible backfill jobs is filtered, selecting only those which will actually fit in the current backfill window.
2. The *degree-of-fit* of each job is determined based on the SCHEDULINGCRITERIA parameter (i.e., processors, seconds, processor-seconds, etc.) (i.e., if processors is selected, the job which requests the most processors will have the best fit)
3. The job with the best fit is started and the backfill window size adjusted.
4. While backfill jobs and idle resources remain, repeat step 1.

Other backfill policies behave in a similar manner with more details available in the Maui documentation.

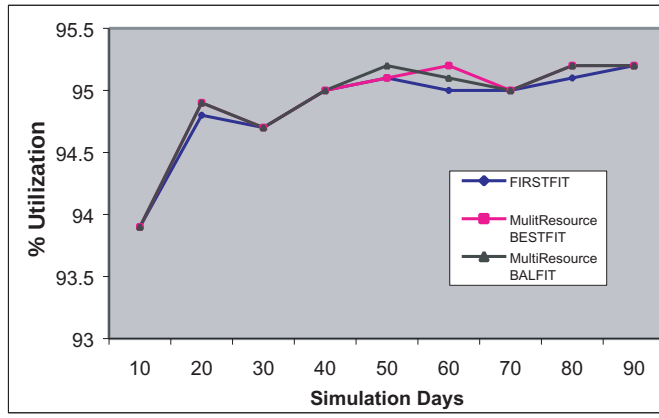

Fig. 4. Comparison of various backfill algorithms.

Figure 4 shows a comparison of backfill algorithms. This graph was generated using the emulation capabilities within the Maui scheduler which have be demonstrated in [7,8,9]. Notice that over the life of the simulation, the resulting utilization for all three algorithms track each other closely; so closely that it doesn't seem to matter which algorithm is chosen. When Maui starts up, priority jobs are scheduled. A backfill round then follows which places all possible jobs on the remaining resources until the space is insufficient to allow any backfill job to run. After this first iteration, Maui can only backfill when a new job is submitted (i.e., it may be small enough to run on available idle resources) or when a running job completes freeing additional resources for scheduling. Scheduling iteration granularity is generally so small that most often only a single job completes or enters the queue in a single iteration. Often, a large percentage of the freed resources are dedicated to a FIFO priority job and are not available for backfill. The reduced set of free resources is rarely adequate to run more than one backfill job. These conditions often result in the backfill algorithms making

the same job selection for backfill. In the cases where more than one job could be run, the algorithms often selected the jobs in different order, but were constrained by resource availability to start the same set of jobs. The cases that allowed more than two jobs to be backfilled within a single iteration allowed the algorithms to differentiate themselves. However, these cases were so infrequent statistically as to have no significant impact on the overall statistics. The algorithms could be reevaluated with very large scheduling intervals to increase job turnover. However, it would not reflect real world conditions as the current results do.

There is one important note. By default, Maui reserves only the highest priority job resulting in a very liberal and aggressive backfill. This reservation guarantees that backfilled jobs will not delay the highest and only the highest priority job. This reservation behavior fails to provide any resource protection for priority jobs other than the first, meaning these jobs could potentially be significantly delayed. However, by minimizing the number of constraints imposed on the scheduler, it allows it more freedom to optimize its schedule, potentially resulting in better overall system utilization and job turnaround time. The parameter RESERVATIONDEPTH is available to control how conservative/liberal the backfill policy is. This parameter controls how deep down the priority queue reservations should be made. A large number for RESERVATIONDEPTH results in conservative backfill behavior. Sites can use this parameter to obtain their desired balance level between priority based job distribution and system utilization.

5 Job Prioritization

Job prioritization is an often overlooked aspect of batch job management. While trivially simple FIFO job prioritization algorithms can satisfy basic needs, much in the way of site policy can be expressed via flexible job prioritization. This allows the site to avoid resorting to an endless array of queues and being affected by the potential resource fragmentation drawbacks associated with them. The Maui prioritization mechanism takes into account 6 main categories of information which are listed in Table 1.

Priority components can be weighted and combined with other scheduling mechanisms to deliver higher overall system utilization, balanced job queue time expansion factors, and prevent job starvation. Priority adjustments are also often used as a mechanism of obtaining quality of service targets for a subset of jobs and for favoring short term resource distribution patterns along job credential and job requirement boundaries.

5.1 Priority Algorithm

Because there are so many factors incorporated into the scheduling decision, with a corresponding number of metrics, (i.e., minutes queued and processors requested) a hierarchy of priority weights is required to allow priority tuning at a sensible level. The high-level priority calculation for job J is as follows:

Table 1. Maui Priority Components

Priority Component	Evaluation Metrics	Use
Service	Current queue time and expansion factor	Allows favoring jobs with lowest current scheduling performance (promotes balanced delivery of job queuetime and expansion factor service levels)
Requested Resources	Requested processors, memory, swap, local disk, nodes, and processor-equivalents	Allows favoring of jobs which meet various requested resource constraints (i.e., favoring large processor jobs counters backfills proclivity for smaller jobs and improves overall system utilization)
Fairshare	User, group, account, QoS, and Class fairshare utilization	Allows favoring jobs based on historical usage associated with their credentials
Direct Priority Specification	User, group, account, QoS, and Class administrator specified priorities	Allows political priorities to be assigned to various groups
Target	Current delta between measured and target queue time and expansion factor values	Allows ability to specify service targets and enable non-linear priority growth to enable a job to reach this service target
Bypass	Job bypass count	Allows favoring of jobs bypassed by backfill to prevent backfill based job starvation

```
Priority = SERVICEWEIGHT * SERVICEFACTOR +
                    RESOURCEWEIGHT * RESOURCEFACTOR +
                    FAIRSHAREWEIGHT * FAIRSHAREFACTOR +
                    DIRECTSPECWEIGHT * DIRECTSPECFACTOR +
                    TARGETWEIGHT * TARGETFACTOR +
                    BYPASSWEIGHT * BYPASSFACTOR
```

where each `*WEIGHT` value is a configurable parameter and each `*FACTOR` component is calculated from subcomponents as described in table 1. Note that the `*CAP` parameters below are also configurable parameters which allow a site to cap the contribution of a particular priority factor.

6 Fairshare

There are a number of interpretations of the term fairshare as applied to batch systems. In general, however, they each involve a mechanism which controls the distribution of delivered resources across various job attribute-based dimensions. They do this by tracking a utilization metric over time and using this historical data to adjust scheduling behavior so as to maintain resource usage within configured fairshare constraints. The above vague description of fairshare leaves great room for interpretation and leaves many algorithmic questions unanswered.

Table 2. Job Priority Component Equations. **NOTE:** XFactor/XF represents expansion factor information calculated as (QueueTime - ExecutionTime / (ExecutionTime)

Factor	**Formula**
Service	QueueTimeWeight * min(QueueTimeCap,QueueTime$_J$) + XFactorWeight * min(XFCap,XFactor$_J$)
Resource	MIN(RESOURCECAP, NODEWEIGHT * Nodes$_J$ + PROCWEIGHT * Processors$_J$ + MEMWEIGHT * Memory$_J$ + SWAPWEIGHT * Swap$_J$ + DISKWEIGHT * Disk$_J$ + PEWEIGHT * PE$_J$)
Fairshare	MIN(FSCAP, FSUSERWEIGHT * FSDeltaUserUsage[User$_J$] + FSGROUPWEIGHT * FSDeltaGroupUsage[Group$_J$] + FSACCOUNTWEIGHT * FSDeltaAccountUsage[Account$_J$] + FSQOSWEIGHT * FSDeltaQOSUsage[QOS$_J$] + FSCLASSWEIGHT * FSDeltaClassUsage[Class$_J$])
Directspec	USERWEIGHT * Priority[User$_J$] + GROUPWEIGHT * Priority[Group$_J$] + ACCOUNTWEIGHT * Priority[Account$_J$] + QOSWEIGHT * Priority[QOS$_J$] + CLASSWEIGHT * Priority[Class$_J$]
Target	(MAX(.0001,XFTarget - XFCurrent$_J$)$^{-2}$ + (MAX(.0001,QTTarget - QTCurrent$_J$)$^{-2}$ NOTE: XF is a unitless ratio while QT is reported in minutes.
Bypass	BypassCount$_J$

For example, it is not clear what the metric of utilization should be nor to which job attributes this correlation data should be correlated. Also, the method of compiling historical data in order to compare it to a particular target value is unclear. Finally, the significant issue of how fairshare targets are enforced is left completely open.

Maui offers flexibility in configuring fairshare in areas including the tracked utilization metric, the utilization to job correlation attributes, the historical period, and the method of fairshare enforcement. Figure 5 shows a typical Maui fairshare configuration.

```
FSPOLICY   PSDEDICATED # track fairshare usage by dedicated proc-seconds
FSINTERVAL    12:00:00 # maintain 12 hour fairshare utilization records
FSDEPTH             14 # track effective usage using last 14 records
FSDECAY           0.80 # decay historical records
FSWEIGHT           100 # specify relative fairshare priority weight
USERFSWEIGHT         2 # relative user fairshare impact
GROUPFSWEIGHT        1 # relative group fairshare impact
QOSFSWEIGHT         10 # relative QOS fairshare impact
CLASSFSWEIGHT        4 # relative class fairshare impact

UserCfg[BigKahuna]    FSTARGET=50    # target usage of 50% (target)
GroupCfg[staff]       FSTARGET=10.0- # target usage below 10% (ceiling)
QOSCfg[HighPriority]  FSTARGET=40.0+ # target usage above 40% (floor)
ClassCfg[interactive] FSTARGET=15.0^ # ignore interactive jobs
                                     # if usage exceeds 15% (cap)
```

Fig. 5. Sample Fairshare Configuration.

Fairshare target usage can be specified on a per user, group, account, QOS, or class basis by way of a fairshare target. Each target is specified as a percentage value where each value is interpreted as a percent of delivered utilization. The use of delivered utilization as the target basis as opposed to using percent of configured or available resources allows the fairshare system to transparently take into account factors such scheduling inefficiencies, system maintenance, job failures, etc.

Fairshare targets can be specified as floors, ceilings, targets, and caps. In the above example, Maui will adjust job priority in an attempt to deliver 50% of delivered processor-hours to user `BigKahuna`, no more than 10% to group staff, and at least 40% to QOS `HighPriority`. The config file also specifies a cap on the class `interactive` instructing Maui to block interactive class jobs from running if the weighted one week usage of the class ever exceeds 15%.

6.1 Fairshare Algorithm

The fairshare algorithm is composed of several parts. These parts handle tasks including the updating of historical fairshare usage information, managing fairshare windows, determining effective fairshare usage, and determining the impact of a job's various effective fairshare usage components.

Updating Historical Fairshare Usage Information. The first issue in a fairshare system is determining the metric of utilization measurement. Likely candidates include utilized cpu and dedicated cpu. The first metric, utilized cpu *charges* a job only for the cpu actually consumed by job processes. The latter, charges a job for all the processing cycles dedicated to the job, regardless of whether or not the job made effective use of them. In a multi-resource, time-sharing, or shared node system, these metrics may not be adequate as they ignore the consumption of non-processor resources. In addition to these CPU metrics, Maui includes an option to track resource consumption by *processor equivalent* metric (PE), where a job's requested PE value is equivalent to

```
PE = MAX(ProcsRequestedByJob / TotalConfiguredProcs,
 MemoryRequestedByJob / TotalConfiguredMemory,
 DiskRequestedByJob / TotalConfiguredDisk,
 SwapRequestedByJob / TotalConfiguredSwap) *
 TotalConfiguredProcs
```

This metric determines a job's most constraining resource consumption and translates it into an equivalent processor count. For example, a 1 processor 4 GB job running on a system with a total of 8 processors and 16 GB of RAM would have a PE of 2 (i.e. MAX(1/8,4/16)*8= 2). To update fairshare usage information, the algorithm steps through the list of active jobs and the list of credentials associated with each job. Typically, each job is associated with a user, group, class (or queue), quality of service (QOS) level, and an optional account. The fairshare usage for each recorded job credential is incremented by the job's fairshare metric amount multiplied by the time interval since the last fairshare measurement was taken as shown below:

```
for (J in JobList)
  for (C in J->CredentialList)
    FSUsage[C->Type][C->Name][0] += <FSMETRIC> * Interval
```

Determining Effective Fairshare Usage. If fairshare targets are to be used, a mechanism for compiling fairshare information collected over time into a single effective usage value is required. This mechanism must determine the timeframe covered and how this information is to be aged. Maui's fairshare algorithm utilizes the concept of fairshare windows each covering a particular period of time. The algorithm allows a site to specify how long each window should last, how fairshare usage in each window should be weighted, and how many windows should be evaluated in obtaining the final effective fairshare usage. For example. a site may wish to make fairshare adjustments based on usage of the previous 8 days. To do this, they may choose to evaluate 8 fairshare windows each consisting of 24 hour periods, with a decay, or aging factor of 0.75 as seen in Figure 6.

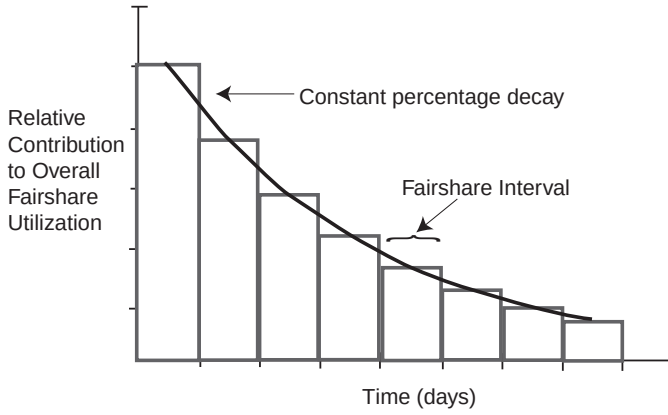

Fig. 6. Effective Fairshare Usage.

To maintain fairshare windows, Maui rolls its fairshare window information each time a fairshare window boundary is reached as shown in the algorithm below:

```
for (N=1->FSDepth)
{
   FSUsage[ObjectType][ObjectName][N] =
      FSUsage[ObjectType][ObjectName][N-1]
}
FSUsage[ObjectType][ObjectName][0] = 0.0;
```

The effective fairshare usage is then calculated at each scheduling algorithm using the following:

```
FSEffectiveUsage[ObjectType][ObjectIndex] = 0.0
for (N=0->FSDEPTH)
  FSEffectiveUsage[ObjectType][ObjectIndex] +=
 FSUsage[ObjectType][ObjectIndex][N] * (FSDECAY ^ N)
```

Determining the Impact of Fairshare Information. Maui utilizes fairshare information in one of two ways. If a fairshare target, floor, or ceiling is specified, fairshare information is used to adjust job priority. If a fairshare cap is specified, fairshare utilization information is used to determine a job's acceptability to be scheduled. (See table 3)

Table 3. Fairshare Target Types

Target Type	**Scheduler Action**
Target	Always adjust job priority to favor target usage
Floor	Increase job priority if usage drops below target
Ceiling	Decrease job priority if usage exceeds target
Cap	Do not consider job for scheduling if usage exceeds target

As mentioned previously, Maui determines percentage fairshare utilization with respect to actual delivered utilization, not configured or available utilization. This is calculated using the following equation:

```
FSPercentUsage[ObjectType][ObjectIndex] =
FSEffectiveUsage[ObjectType][ObjectIndex] /
FSTotalEffectiveUsage
```

The impact of all relevant fairshare targets are considered and incorporated into the final priority adjustment of a job as described in section 3.

There is a common misperception about fairshare. Some sites initially believe that they can specify fairshare targets and that the scheduler can force these targets to be met. This is not the case. Since a fairshare system cannot control the mix of jobs submitted, it cannot guarantee successful fulfillment of targets. If a high target user does not submit jobs, then his target cannot be met regardless of how hard the scheduler tries and preventing other jobs from running will not help. The purpose of a fairshare system should be to *steer* existing workload, favoring jobs below the target so as to improve the turnaround time of these jobs and perhaps allow the associated users the opportunity to submit subsequent dependent jobs sooner. A fairshare system can only *push* submitted jobs so as to approach targets, hence the extensive use of priority adjustments.

The Maui fairshare system fits neatly in a time-based spectrum of resource distribution capabilities. At the short term end, a number of throttling policies are available allowing a specification of how many jobs, processors, nodes, etc. can be used by a given entity at a single time (i.e., the sum of processors simultaneously utilized by user John's jobs may not exceed 32). Fairshare allows resource usage targets to be specified over a given time frame, generally a

few days to a few weeks. For longer time frames, Maui interfaces to powerful allocation management systems, such as PNNL's QBank, which allow per user allocations to be managed over an arbitrary time frame. Such systems allow Maui to check the *available balance* of a job, blocking those jobs with inadequate balances, and debiting allocations for successfully completed jobs.

6.2 Summary

The purpose of this paper was to present the backfill, job prioritization, and fairshare algorithms used within the Maui scheduler. While wide-spread use and numerous informal evaluations of the scheduler have demonstrated value in these algorithms, no formal or exhaustive analysis of the effectiveness of each algorithm has been published. These algorithms will be evaluated individually in forthcoming papers.

While the described backfill, priority, and fairshare systems appear to have met the needs of a wide spectrum of HPC architectures and site policies, research and development in these areas continue. Significant enhancements to Maui also continue in the realm of quality of service delivery and new preemption based backfill optimizations. Additional work is also ongoing in extending Maui's existing interface for grid applications and general metascheduling, with a near term focus on improving job start time estimations. Research regarding the effect of the quality of this start time information on the performance of multi-system load balancing systems is currently underway.

References

1. D. Jackson. The Maui Scheduler. Technical report. http://supercluster.org/projects/maui.
2. J.S. Skovira, W. Chen, and H. Zhou. The EASY - LoadLeveler API Project. *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science 1162*, pages 41–47, 1996.
3. R.L. Henderson. Job scheduling under the Portable Batch System. *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science*, 949, 1995.
4. J.M. Barton and N. Bitar. A scalable multi-discipline multiple processor scheduling framework for IRIX. *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science*, 949, 1995.
5. D. Jackson. HPC workload repository. Technical report. http://www.supercluster.org/research/traces.
6. D. Feitelson and A. Mu'alem Weil. Utilization and predicability in scheduling the IBM SP2 with backfilling. In *Proceedings of IPPS/SPDP*, April 1998.
7. Q. Snell, M. Clement, D. Jackson, and C. Gregory. The performance impact of advance reservation metascheduling. *Lecture Notes in Computer Science: Job Scheduling Strategiew for Parallel Processing*, 1911, 2000.
8. John Jardine. Avoiding livelock using the Y Metascheduler and exponential backoff. Master's thesis, Brigham Young University, 2000.
9. D. Jackson, Q. Snell, and M. Clement. Simulation based HPC workload analysis. In *International Parallel and Distributed Processing Symposium*, 2001.