# The UNICOS Fair Share Scheduler as a Feedback Control System

*Cass Everitt* and *Terry Jones*, Grumman Data Systems and Services, Inc., and *Robert M. Knesel*, Naval Oceanographic Office (NAVOCEANO)

**ABSTRACT:** *The UNICOS Fair Share Scheduler is a departure from the conventional UNICOS per-process scheduler in that it distributes computational resources to users or accounts rather than directly to processes. With its hierarchical implementation, it provides the ability to accommodate political goals at multiple levels of organization. This hierarchy, or share tree, adds greater functionality but increases complexity. The complexity introduced by the hierarchy often makes it difficult to understand and predict how Fair Share will behave under a variety of conditions. Analyzing the Fair Share Scheduler as a feedback control system provides some visual insight as to how it should behave and some possible causes for unexpected behavior.*

## 1 Introduction

The Naval Oceanographic Office (NAVOCEANO) Department of Defense (DoD) Major Shared Resource Center (MSRC) was established under the DoD High-Performance Computing Modernization Plan (HPCMP) in October 1994 as the second such facility of its kind. The MSRC was created from the computational assets previously dedicated to NAVOCEANO to support the Primary Oceanographic Prediction System (POPS). With the designation of the former NAVOCEANO POPS facility as a DoD MSRC, the facility became available to all DoD Research and Development activities requiring high-performance computing (HPC) resources. Located at Stennis Space Center, Mississippi, the existing assets were augmented with a Cray C916/16-1024 system to support the increased workload resulting from the expansion of the user community that came about with the designation as a DoD-wide resource. The UNICOS Fair Share Scheduler, already in use at the other existing DoD MSRC, was utilized from the beginning as a means of accomplishing the allocations of CPU time between the multiple DoD R&D groups who would use the systems.

User concerns about accounting report discrepancies versus fair share entitlements on the NAVOCEANO DoD Major Shared Resource C916/16-1024 system led to a detailed investigation to determine the reasons and the remedy for them. Specifically, when the monthly accounting reports were examined, the users noticed that significant differences existed between the actual utilization and the expected utilization for various groups considering what was understood about Fair Share Scheduler behavior. The investigation was wide ranging and uncovered numerous significant issues which govern system throughput and explain the differences seen between the accounting reports and the entitlement definitions. As will be shown in this paper, some of these issues can be attributed to a disparity between how a Fair Share Scheduler should intuitively function and how it actually performs. Other issues deal with the expectation that a CPU scheduler will be able to closely match a target allocation scheme across all levels of allocation granularity when it has limited ability to consider and affect allocations of other system resources.

### 1.1 Problem Statement

The problem posed was to develop an understanding of why these discrepancies existed when Fair Share Scheduler was being employed to prevent them. Users were demanding to know why they weren't "getting their fair share."

The intuitive behavior of a Fair Share Scheduler operating with a hierarchical allocation scheme (multiple levels of subdivision of CPU time) is that it will act to limit overconsumers and boost underconsumers so that all levels of the allocation receive exactly the entitlement of CPU time defined. The expectation is that no one group which has sufficient workload will receive less than the allocation defined for them. This applies to all groups in the hierarchy, no matter what level. Thus, the sum of CPU time accumulated for all subgroups of a given group will be no less than the amount of CPU time allocated to that group. This produces a corollary expectation that surplus CPU time from subgroups with insufficient workload will be reallocated proportionally among the subgroups with sufficient workload. Surplus time is produced when a subgroup's actual usage is less than its entitlement. The Fair Share Scheduler is expected to force the usage levels for all groups to converge on the defined allocations within a given accounting interval (temporal behavior). Thus, correct allocation and functionality is measured by system resource accounting reports that describe actual CPU usage. However, the Fair Share Scheduler actually behaves in an instantaneous fashion, and while convergence is often observed, it is seldom monotonic and complete.

### *1.2 Reasons for Using the Fair Share Scheduler*

The DoD HPCMP has been created to modernize the total high-performance computational capability of the military research, development, test, and evaluation (RDT&E) community to levels comparable to that available in the foremost civilian and other Government agencies, such as the National Science Foundation (NSF) sites. Economic necessities of the post-Cold War environment have reduced total available resources for DDR&E activities and resulted in the establishment of high-performance computational facilities shared by the entire DoD RDT&E have community. This consolidation of resources generates a diverse widely geographically dispersed user community which has previously utilized local resources typically dedicated to a single branch or organization within a single branch of the DoD [5]. Specifically, the user community created presents a wide range of computational requirements to each computational platform. With funding for the shared facilities originating from the parent DoD organization, a means of equitable distribution of the computational resources among the entire DoD RDT&E community is required.

NAVOCEANO has a continuing requirement to provide near-real-time support with the POPS oceanographic products delivered on a daily schedule to operational Navy fleet units deployed world-wide. This necessitates that a portion (up to 15%) of the computational resources remain available for NAVOCEANO's internal use in generating the products, remedial support of the models, and continuing development efforts to support future requirements. Thus NAVOCEANO requires a separate entitlement, and within that entitlement, two priority levels of workload exist, each with their own separate entitlements reflecting their relative importance: the oceanographic products that must be produced within specific time windows and the remedial and future requirement support.

### *1.3 NAVOCEANO MSRC Share Tree and FSS Parameters*

Two models of the fair share scheduler exist: a flat, or single-level model and a hierarchical model. The flat model divides the system allocations one time. The hierarchical model allows for the situation where independent organizations share a system and that the user shares of the system depend on the allocation of their organization's entitlement and their entitlement within their organization [6]. While more complex, this model is preferred for most environments. It provides an easy means to allocate shares as each organization is free to allocate its shares as if it had exclusive use of the system. This multiple independent organization scheme is inherent in the NAVOCEANO DoD MSRC resource allocation scheme and the HPCMP directions in general.

The NAVOCEANO DoD MSRC configuration of Fair Share Scheduler uses the Share by Account and the Adjust Groups features of UNICOS Fair Share Scheduler.

The Adjust Groups feature is designed to redistribute the resources unused by groups to other groups with the same parent. This is to prevent system resources from going unused when not all groups are active. There are significant implications of this feature which are explored in depth in Section 2.0, Investigation of the Fair Share Scheduler.

Share by Account calculates usage and adjusts priorities for each active account ID (acid). The NAVOCEANO MSRC user environment supports individual users working on multiple projects, with separate accounting for each project (Project-Level Accounting). Users supporting multiple projects typically employ a single userid and switch between projects using the **newacct(1)** command. This allows them to work under a different set of fair-share priorities for each project and assures that CPU utilization accrued under one project is not applied to the user's other, independent projects. Implementation of Share By Account is effected by creating shareholder nodes (the terminal nodes in the resource group tree) for each project and assigning share allocations to them. Userids are assigned a default and permitted list of acids in their User Data Base (UDB) entry which correspond to the shareholder nodes [3]. In all other respects, Fair Share Scheduler works as if it were accumulating usage for userids.

The requirements from Section 1.2 are combined to produce the basic Resource Group Tree illustrated in Figure 1-1, NAVOCEANO DoD MSRC Fair Share Scheduler Hierarchy. The shareholders (project IDs) are connected to the lowest level resource group in the share hierarchy; and for the majority of the cases, are all assigned equal shares. In the NAVOCEANO DoD MSRC configuration, this is typically a department or a lab of a DoD Service Branch. These lowest levels of resource groups are then connected to their parent organization and are assigned shares to effect the suballocation of the parent organization's entitlement. This process continues in turn at each successively higher level of organization until the parent DoD Service (Army, Navy R&D, Air Force, etc.) is reached. The parent organization for the Service Branches has a single sibling in the resource group tree, NAVOCEANO. These two groups have the User resource group as their parent node and are assigned shares to implement the allocation of 15% of the system resources to NAVOCEANO. The "System" and "Staff" resource groups are overallocated with 12.5% each. Neither of these groups are expected to require CPU resources of this magnitude; however, these levels provide for ample "reserve" in the event that a high priority system process or support function is required. Actual combined utilization for these resource groups is typically under 5% combined. By defining 12.5% for each group, Fair Share

Scheduler will see them as underutilized and boost the priority of their processes. The same methodology is used to insure that the POPS Oceanographic Products are run within the requisite daily real-time windows.

### *1.4 System Utilization Inconsistencies Encountered*

Users noticed that the system failed to equitably divide "surplus shares" between two equally entitled major groups. Specifically, attention was focused on the system share realized between the Army, Navy R&D, and Air Force Groups. Reference Figure 1-1, NAVO DoD MSRC Fair Share Scheduler Hierarchy, for the resource groups and their relationships. Both "Army" and "DoD Other" were light users of the system during the period in question, undersaturating their node. With insufficient workload from both groups to utilize their full entitlements, and the FSS ADJGROUPS feature in effect, the amount of "Army" and "DoD Other" entitlements not used (surplus entitlement) should have been distributed equally between the Navy R&D and Air Force resource groups. Both "Navy R&D" and "Air Force" oversaturated their nodes and thus, both should have had roughly the same utilization for the report period in question.

In fact, what actually occurred is presented in Table 1-1, May 1995 NAVOCEANO DoD MSRC C916 Utilization.

Air Force users received 46.78% of the total CPU time in 28646 runs; while Navy R&D users received 35.12% of the total CPU time in 22920 runs. Both groups executed predominately batch runs equally across the batch queue structure. If FSS was performing as required, both groups should have received equal amounts of the system. Table 1-2, Observed Utilization Anomalies illustrates the unexpected results.

Table 1-1. May 1995 NAVOCEANO DoD MSRC C916 Utilization

| Resource Group | CPU Hours | % | No. Jobs | No. Users |
|---|---|---|---|---|
| Army | **442.4701** | **4.19** | **685** | **29** |
| Navy R&D | **3705.8881** | **35.12** | **22920** | **345** |
| Air Force | **4935.7895** | **46.78** | **28646** | **177** |
| DoD Other | **88.9370** | **0.84** | **528** | **13** |
| NAVO | **1158.1426** | **10.98** | **116954** | **121** |
| Support | **219.8616** | **2.08** | **42897** | **45** |

The surpluses from each level were applied among active siblings and propagated down into the tree to determine the actual surplus available for each lower level resource group. This was done in turn for each level to arrive at the actual surplus available for the lack of activity on all other nodes at the same and higher levels. When this was done, the resulting target real entitlement for "Navy R&D" and "Air Force" for this accounting interval was 40.952%. Thus, "Air Force" received 5.83% above their target entitlement and "Navy R&D" received 5.83% below their target entitlement. Clearly something was operating beneath the surface of FSS which required further investigation.

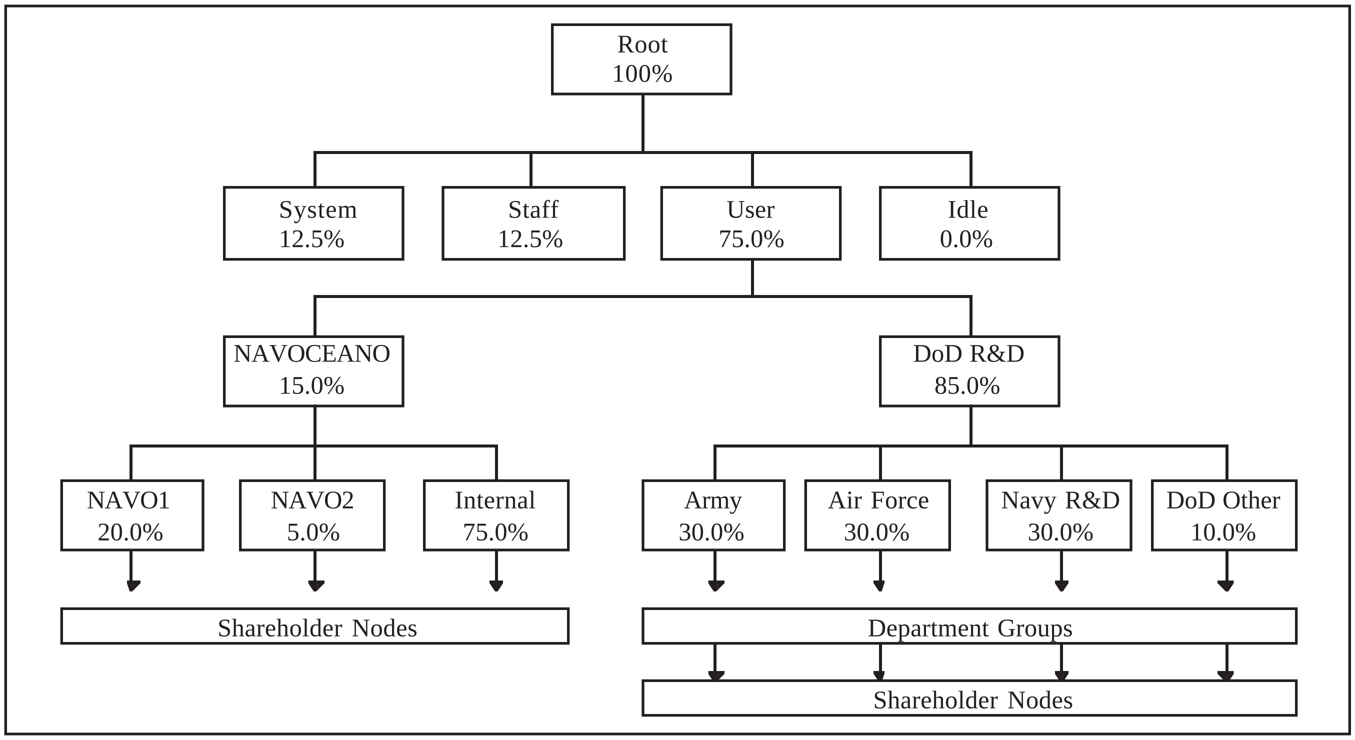


**Figure 1-1: NAVOCEANO DoD MSRC Fair Share Scheduler.**

Table 1-2. Observed Utilization Anomalies

| Resource Group | System Share Entitlement | Actual Utilization | Surplus (Deficit) |
|---|---|---|---|
| Army | **19.125** | **4.19** | **14.935** |
| Navy R&D | **19.125** | **35.12** | **(-15.995)** |
| Air Force | **19.125** | **46.78** | **(-27.655)** |
| DoD Other | **6.375** | **0.84** | **5.535** |
| NAVO | **11.250** | **10.98** | **0.27** |
| Support | **25.000** | **2.08** | **22.92** |

## *1.5 Attempts to Resolve Inconsistencies Using FSS Parameters*

Observations revealed an active user under the Air Force resource group whose jobs, running in a favored queue, were executing with a base *nice* value significantly lower than that of other users. This user's UDB nice[b] value was set to 0, whereas the normal setting is 8. This allowed this user's batch jobs to execute with a base nice value eight points below all other users in the system. Analysis of the process accounting records showed that this user accounted for the majority of the 5% difference noted. When these nice values were corrected, a week-by-week spot check of utilization by the DoD Service branches and the subsequent month's accounting reports showed greater convergence. However, this raised further questions about the internals of FSS and called into question its validity. The question posed was that even if this user's nice value gave him a higher execution priority, why did FSS not still limit his activity so that the Air Force resource group used only its target entitlement?

Moreover, monitoring of the system activity showed a periodicity in the decayed usage for resource groups which closely followed an approximately 24- hour interval. Since activity was relatively constant for these resource groups, decayed usage should also remain fairly constant. Decay seemed to occur normally for groups which had no activity for a period of time, but the 24-hour variability occurred for the consistently oversaturated nodes.

The usage decay half-life during that accounting interval was 8 hours. With three half-lives in a day, usage was decaying too rapidly. This appeared to be involved in the discrepancies.

A full analysis of Fair Share Scheduler was then undertaken to learn the exact nature of the underlying mechanisms. Particular attention was paid to determining reasons why the workload, controlled by Fair Share Scheduler, would fail to achieve reasonable convergence to the target utilization as defined by the entitlements in the share group hierarchy. Once it was ascertained that the Fair Share Scheduler code was at the latest Bugfix levels during the report interval in question, analysis concentrated on the underlying algorithms.

The usage decay half-life was extended in a series of steps to 168 hours (1 week) to coincide more with the accounting interval of one month. Further, this extension was called for in light of the initial results of the analysis of Fair Share Scheduler presented in Section 2.0, Investigation of the Fair Share Scheduler.

Node saturation was also an issue. The NAVOCEANO resource group was failing to reach its full entitlement, in spite of a backlog demand which would easily exceed it. Analysis revealed that these jobs were being caught in backlog and the throughput was simply not high enough to experience node saturation. The problem was more complex and eventually resulted in several URM and general tuning changes. Specifically, the URM ranking algorithm was changed to consider the Fair Share usage information provided to URM (rmgr: /urm/usage_wt) as the most significant factor in ranking jobs for execution. Previously, the age of the job in the input queue (rmgr: /urm/age_wt) was considered the most important [3]. With throughput increased and usage history information considered, more NAVOCEANO jobs were able to be gated into execution The probability of a job entering execution during a specific time interval is inversely proportional to the memory requirements for the job. NAVOCEANO jobs fell into a region outside URM's preferred job size.

During the analysis of the implementation and the requirements of the product, our site developed a model of our own about what a FSS should do. Thus, a model began to emerge which strongly suggested a structure best described using control theory. The requirement for the product resembled a feedback-controlled pulse frequency modulator which evolved into the model presented in this paper. As will be shown, such a model is very applicable in describing a new implementation. The format of this model, when used to analyze the existing implementation, very quickly exposed the algorithmic deficiencies as well as the modifications required to remedy them.

When control theory is applied to the existing implementation and the original flat model is considered, it strongly suggests that the FSS product is actually an empirical extension to the flat model. This is suggested by the existence of the adjust groups flag (shradmin ADJGROUPS) in the UNICOS FSS. With ADJGROUPS disabled, the implementation collapses to the flat model. The ADJGROUPS functionality is implicit in a hierarchical model. This becomes the main reason why FSS fails to perform as required in a complex hierarchical environment. An empirical extension typically does not consider nor anticipate all potential situations under which it is required to perform. FSS is a control mechanism for an extremely complex system and thus the most appropriate means for designing such a control system is by the use of formal control theory. Formal control theory contains the elements necessary to state and design complex control mechanisms succinctly for complex systems.

## 2 Investigation of the Fair Share Scheduler

The UNICOS Fair Share Scheduler operates on the paradigm of share allocation. Each user is given CPU entitlement based on share holdings. Functionally, the user's shares are divided by the system total to define normalized share or system percentage [8]. This normalized share is the magic number that the user will be looking for in accounting reports. The share administrator must define other parameters commensurate to user work load periodicity.

### 2.1 *The Simple (Flat) Fair Share Scheduler*

The original implementations of Fair Share Schedulers assigned shares to users only. This was a great improvement over no share at all in that users with a single process would not be starved by users with multiple processes, and sporadic users would receive high priority during loaded conditions. Likewise, users with more shares received consistently better response.

#### *2.1.1 Modification to the Priority Equation*

Due to history or ease of implementation, the standard decaying priority-based CPU scheduler is used as the foundation for FSS and the model from which all subsequent equations are derived. This model is simply a queue of runnable processes ordered by priority. The algorithm for scheduling is as follows:

**Note:** pri is the number that defines priority where high priority implies low pri and low priority implies high pri.

a) Select process with lowest pri (highest priority) and execute for one OS cycle

b) Increase pri (decrease priority) for that process

c) decay pri (increment priority) for all waiting runnable processes

d) goto a

This mechanism is analogous to a conveyer belt with a CPU at the front. The front process is grabbed by the CPU and operated on (a), then placed at the back of the conveyer (b), while the conveyer moves steadily forward (c).

This old "process fair" CPU scheduler becomes a "user fair" scheduler with a very simple modification to the priority degradation equation (b). The equation must insert the user's process(es) into the queue in such a way that the CPU time spent on that user's work is equal to the user's normalized share. That is, rather than appending a process to the end of the conveyer belt, it is placed closer to the front of the belt as a function of normalized share. This new priority equation is

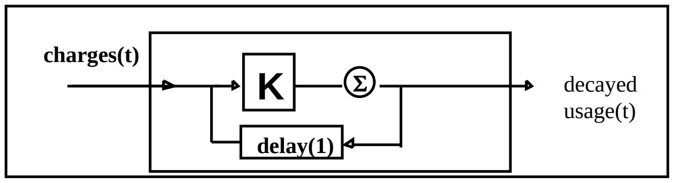


**Figure 2-1: Usage Decay.**

$$\text{pri} = \text{pri} + e^{((1-\text{normalized\_share}(\text{user}))*\text{number\_procs}(\text{user}))} \qquad (1)$$

where number_procs is the number of processes owned by that user that are currently in the run queue.

The computational overhead of performing an exponential operation at every scheduling event may be impractical, but it is required due to the exponential decay mechanism of the scheduler (conveyer). Over an implicit time interval (defined by OS hertz and the length of the run queue - usually a few seconds), this equation (1) will fairly distribute cpu to runnable processes, but aside from being expensive, it does not address the need for fairness over hours or days. This equation can work only with what is currently in the run queue, without regard for users that aren't running at least one process constantly.

#### *2.1.2 Decayed Usage for Memory (Temporal Feedback)*

Because workloads usually have a certain chaotic periodicity, instantaneous fair share is not really the desired goal. In order to accommodate the desire to have system accounting reports resemble share allocations, the FSS must remember past usage. As usage accumulates, the ratio of usage to share should be the same for each user. A factor of usage/share in the priority equation will degrade the priority (increase pri) for users receiving more than their share and increase priority (decrease pri) for users receiving less than their share.

The distinction of a feedback control system is that it doesn't require an exact equation - it is self-correcting. We eliminate the need for an exponential function to define priority, because if we are not exact in our approximation, usage/share will adjust itself to compensate [7]. We can therefore state the fair share priority equation as

$$\begin{aligned}\text{pri} = {} & \text{pri} + (\text{usage}(\text{user})/\text{share}(\text{user})) \\ & \times (\text{num\_procs}(\text{user})/\text{share}(\text{user}))\end{aligned} \qquad (2)$$

which is computationally much simpler than an exponential, and it is fair over time. A decay factor is placed on the usage term in order to limit the obligation to be fair over an infinite interval. For example, suppose share was implemented on a system without decay of usage, and users A and B were each entitled to 50% of the machine. Further suppose that A had daily requirements for system utilization and B had no requirements at all. Six months later, however, B had a project that called for heavy utilization. Without decayed usage, group B will dominate (well over their 50%) for several months in order to compensate for their previous lack of usage, and group A will be unable to meet their daily computational requirements. Providing a decay half-life for usage allows the share administrator to define an effective interval in which machine share must be asked for or lost. This decay also bounds the decayed usage parameter, which is convenient for limited precision machines.

At this point, the primary concern of the FSS is effectively adjusting process priority so that cpu scheduling is fair. With all the complex internals of FSS, the only real transient input and output are charges and pri, respectively. In order to build a representative block diagram model of the FSS feedback control system, we need to simplify some of the operations into functional groupings. The first candidate for grouping is the usage decay block to implement the integral equation for decay (6).

When discussing decay factors, we have several different variables (σ, half-life, K), so to avoid confusion, we will expose their relationship here. Generally, usage decay is specified in half-life, but is only used to define internal variables s and K. The fundamental decay equation is simply

$$f(t) = e^{-\sigma t} \tag{3}$$

Initially, we specify the decay in terms of half-life, which gives us σ, by

$$f(t) = e^{-\sigma t_{\frac{1}{2}}} = 0.5$$
$$\sigma = -\left(\frac{\ln(5)}{t_{\frac{1}{2}}}\right) \tag{4}$$

where t_half is typically specified in hours. Now, we can determine an appropriate decay factor for any time interval by

$$f(t) = e^{-\sigma(tn - t(n-1))} \tag{5}$$

which is especially useful since the FSS delta, Δ, (shradmin -R) is a constant. Since this number is generally specified in seconds, we can define K as:

$$K = e^{-\sigma\frac{\Delta}{3600}} \tag{6}$$

K is then the number (very slightly less than 1.0) displayed as DecayUsage with the *shconsts* option for shrview(1). Since usage is simply the integral of charges over time, it can be written as:

$$\text{usage}(t) = \int_0^\infty \text{charges}(t)\,dt \tag{7}$$

and decayed usage, then, as

$$\text{decayed usage} = \int_0^\infty \text{charges}(t)e^{-\sigma t}\,dt \tag{8}$$

Having come up with a means for calculating decayed usage and defining a control block that provides that functionality, we can now show a block diagram for the flat FSS. Note that the FSS only sets priority and receives charges. The conventional CPU scheduler is still the workhorse.

### *2.2 The Hierarchical Fair Share Scheduler*

The original desire for a Fair Share Scheduler was to provide a mechanism fair to users, not processes; having achieved that, the next requirement is obviously fairness to groups of users, and groups of groups. Stated simply, the new goal is hierarchical FSS.

The flat FSS is essentially a very simple implementation in that each node is functionally independent. Figure 2-1, Flat Fair Share Scheduler Model, shows that there is no need for communication between nodes. This is essential, however, for hierarchical share for satisfying the requirement of fairness to groups. For example, consider organizations A and B that each have equal entitlements, and each of the users within the organization have equal entitlements. If two users of organization A are active, and one user from organization B is active, without compensation, organization A will receive 2/3 share, and organization B will receive 1/3 share. In order for organization B to receive its entitlement, an adjustment must be made to alter the priority of the organization B user. In order to sense this error in proportion, nodes must be maintained at the organizational level and fed by the charges incurred by its children.

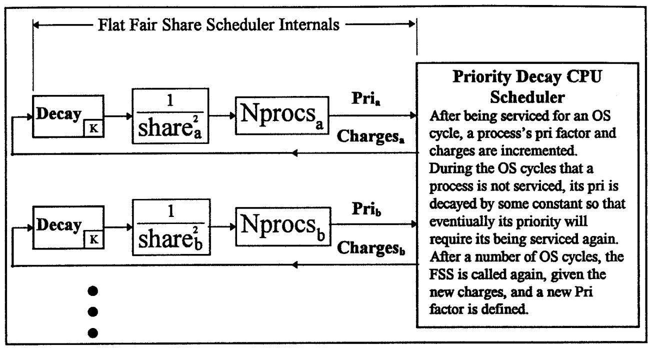
#### *2.2.1 Current FSS Approach*

The hierarchical implementation to FSS is much more complicated that its flat predecessor. Part of this complexity is intrinsic to the goal, but some may be unnecessary. Essentially, the flat portion of the algorithm is unchanged, but the method of defining shares is specified in hierarchical terms. For example, suppose that there are two organizations, A and B, and each hold 200 shares. Their effective machine share is 50%. Suppose also that organization A has two users, and each holds 150 shares. Their effective group shares are 50%, and their effective machine shares are 25%.

This mechanism can be used to collapse any tree hierarchy into a flat share. In fact, without the ADJGROUPS flag set, that is exactly how share operates. In practice, it makes little sense to define a share tree without telling share to use it, so ADJGROUPS is usually set.

##### *2.2.1.1 Adjusting Groups Complex*

The adjust groups algorithm is a separate module within the FSS. Its function is to define a weighting parameter for each terminal node that is used to calculate the Pri parameter. The caveats of the hierarchical implementation reside almost exclusively in this adjust groups algorithm, and its method of providing fairness.



After being serviced for an OS cycle, a process's pri factor and charges are incremented. During the OS cycles that a process is not serviced, its pri is decayed by some constant so that eventiually its priority will require its being serviced again. After a number of OS cycles, the FSS is called again, given the new charges, and a new Pri factor is defined.

**Figure 2-2: Flat Fair Share Scheduler Model.**

The method of adjusting groups takes a fundamental step away from the proven paradigm of decayed usage, by determining fairness among groups with short-term (scheduler delta) charges [6]. As was stated in the discussion of the flat FSS, instantaneous FSS, while instantaneously accurate, does nothing to satisfy the long-term requirements for fairness. The analogies are not direct from flat to hierarchical, but the result is the same. It requires more algorithmic and hence computational overhead to produce this instantaneous fairness to groups than a temporal FSS.

In terms of predictability, having components of the scheduler that are transient (group adjustment) and temporal (flat component) tend to produce unanticipated results. The transient element allows the chaotic periodicity associated with the user loads to become part of its output rather than forcing a smoothing function.

### *2.2.1.2 Limitations of the current approach*

There are many limitations to the FSS that are imposed by the environment in which it operates; regardless of implementation limitations, they will still cause the data to be skewed. These concerns are elaborated on later, but part of the problem in determining algorithmic limitations is attempting to filter out the environmental factors.

From a strictly algorithmic analysis, there are scenarios that raise questions not answered by the current implementation. Consider the share hierarchy shown in Figure 2-3. There are two organizations named A and B: A has users 1 and 2; B has users 3 and 4. All entitlements are divided equally. With ADJGROUPS in operation, suppose 4 runs for a sustained period. Later, 1 and 2 log in and begin running. Since their usages are low, they become the dominant jobs on the system. Suppose, then 4's process terminates and 3 logs in and begins running. How does 3's process perform? Since group A has still not received its fair share because of 4's long-running process, we might expect that 3 will run poorly. This would be fair to A, but unfair to 3. As organizational utilization constitutes a larger workload than per user utilization, it makes sense to adjust the organizational discrepancy of A and B first, then deal with the discrepancy between 3 and 4. In reality, the fairness between groups is governed by transient analysis, so 3's process becomes dominant. This is unfair to A.

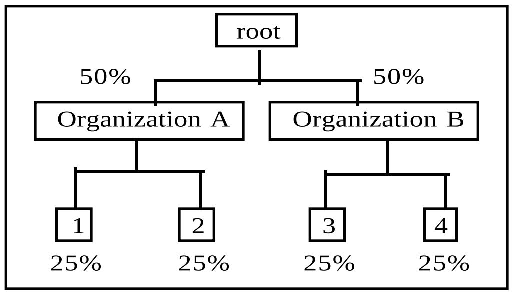


**Figure 2-3: Share tree example**

Under consistent loading conditions by all users and a very long usage decay half-life, this problem becomes less apparent, but it still induces a bound on the accuracy of the FSS. For that reason, small share allocations and deep share trees are not recommended.

### *2.2.2 Fully Hierarchical Feedback Approach*

In analyzing the flat FSS model it can be seen that there is no need for node communication. Unfortunately, this is not the case for the extension to a hierarchical FSS. The flat fair share scheduler has need for only one datum: the current usage. The data needs are greater with a hierarchical scheduler. Before, priority was adjusted directly because the only concern was for a particular node to receive its fair share. Now we must consider a node as a component of a larger resource group. The share mechanism must work to realize its entitlement among its siblings, but only within the limits of the parent's entitlements. In the flat case, the share mechanism could be run directly from usage, because the parent node was the root node and the children could get no more than the machine's capability. For the hierarchical FSS to work in any sort of intuitive way, parent nodes must be able to place a limit on group consumption. This need is not addressed in the flat FSS because the only parent node is the root node and it already has an intrinsic limitation. For this reason, using the existing flat FSS algorithms is impossible, and a new technique must be devised.

There are really only two fundamental concerns when examining any branch point of a hierarchical FSS. First, the node must receive its entitlement. Second, the node must fairly distribute its resources. For this reason the duties of our proposed hierarchical FSS are broken into two administrative tasks. One duty is for the share node to act as the advocate for its offspring, the other is for the share redistribution node to act as the arbitrator to its offspring. The first functional block (the share node) determines whether the entitlements for its group are being realized and adjusts itself to do so. The second block (the share redistribution block) periodically redefines entitlement based on decayed usage. This structure allows for fairness to the group via the share node as well as fairness among the group members via the share redistribution node.

### *2.2.2.1 The Share Node*

The share node shown in Figure 2-4 is the fundamental feedback control system in this structure. Its primary inputs are base_share, which is the parent's share, and transient_share which is the node's current share relative to its siblings. The product of these two numbers defines the actual share for that node. The feedback input for the share node is simply the charges accrued by that node over the last scheduler delta. The output uses all of these data to generate a base_share for its children. Internally the share node has two major components: a low pass filter block, and an error block.

The low pass filter shown in Figure 2-5 is used to filter out high frequency noise in the charges value to stabilize the input to the error block. The low pass filter has one parameter, $K_c$,

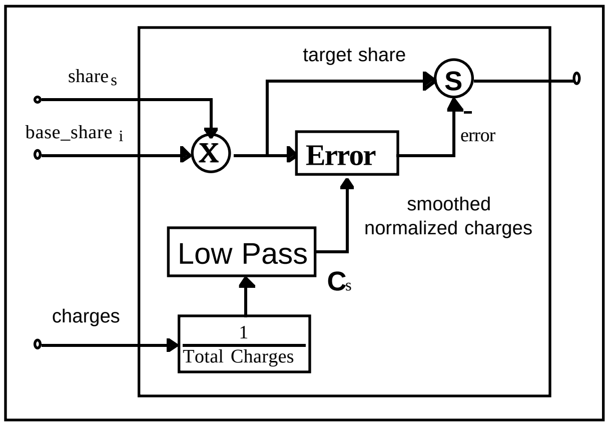


Figure 2-4: Share Node.

with range (0.0, 1.0], which defines its transient response. A $K_c$ of 1.0 is a unity (all-pass) filter, while values close to 0.0 pass only low frequency variations. The electrical system analogy is that of an RC network where *charges* represents the voltage source, $K_c$ represents the resistance (R), and the output is the voltage across the capacitor (C). This filter allows the scheduler to adapt to abrupt changes in user workload without burdening it with unstable input.

We then apply the smoothed charges function to the error block shown in Figure 2-6. The other input to the error block is the desired share. The difference between these two numbers is multiplied by another scaling factor, $K_\mu$, which defines the speed with which priority adjustments are made. This output, defined as the *error*, is then subtracted from the node's actual share to define the base_share of its children.

The only configurable parameters of the share node (aside from share tree defined entitlements) are the two constants, $K_c$ and $K_\mu$. $K_c$ should be set to define the length of time considered when determining error of entitlements versus realized utilization. This window should probably be on the order of one to five minutes. This value can be the same for all share nodes. A little more care must be taken when defining the parameter $K_\mu$. Realistically the operating range for this parameter should be [0.0, 2.0], where values close to 0.0 produce an overdamped convergence to the appropriate share (0.0 produces no error correction at all), and larger values produce varying degrees of overshoot,

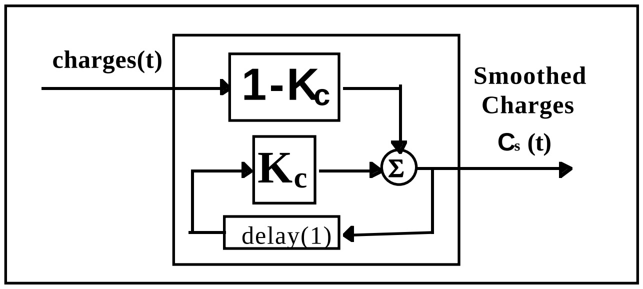


Figure 2-5: Low Pass Filter.

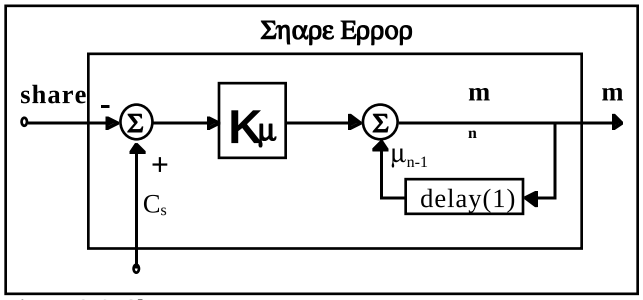


Figure 2-6: Share error.

but faster convergence. The setting of this parameter must also consider the depth of the node within the share tree. The parent nodes must give their children a little time to correct to their adjustments before they take action themselves, so $K_\mu$ should increase proportional to node depth.

### 2.2.2.2 *Share Redistribution Node*

The share redistribution node provides the dynamic distribution of share among siblings based on their decayed usage and their entitlement. We define a functional block, shown in Figure 2-7, for providing this utility, but for simplicity's sake we define its function mathematically and graphically.

The mathematics used to redistribute shares are really just a position control system in n-dimensional space. We will consider the situation where n = 2 (that is, a parent node with two children) for diagramming purposes, but the idea applies for any n.

First, we consider our two-dimensional space of Figure 2-8. Each child node, A and B, has an axis orthogonal to its sibling (the x and y axes in this case), and the coordinates of the current "position" are defined by the decayed usage of each child (decayed usage(A), decayed usage(B)). There is also a target line defined by

$$x/\text{entitlement}(A) = y/\text{entitlement}(B). \qquad (9)$$

If the resources of this node are being fairly divided among the children, the current position will be on this line. If they are not, then we need to define a new vector (share distribution) that

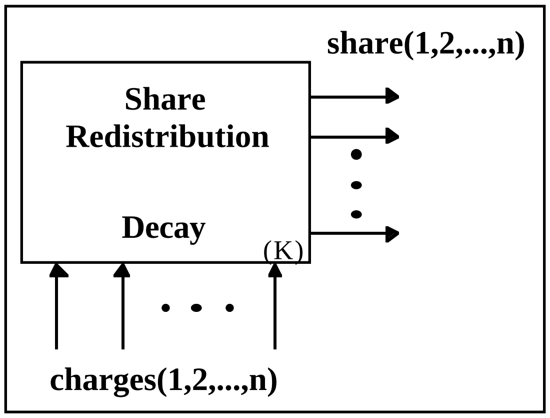


Figure 2-7: Share Redistribution Function.

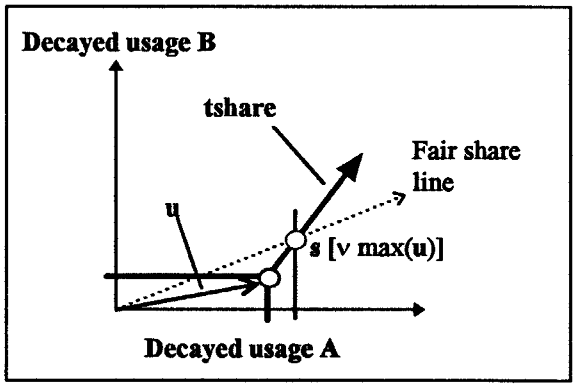


Figure 2-8: Share Redistribution Behavior.

will lead us to fair distribution over time. The equation to satisfy this need is

$$\mathbf{tshare} = s \times [v \times \max(\mathbf{u})] - \mathbf{u} \qquad (10)$$

where **tshare** is the transient share vector, n is the scaling factor that defines the point on the fair share line that is targeted, and max(**u**) is the largest component of the decayed usage vector. **S** is the entitlement vector and **u** is the decayed usage vector (generated by the **charges** vector sent through n decay blocks). **Tshare** is then normalized to define a share percentage for each child.

The parameter n is of importance in that it will allow the overconsumer a small portion of the parent's share. This keeps from marooning overconsumers.

The share redistribution process is not a very expensive operation, but it need not be performed at every scheduler delta either. For example if you have a usage decay half life of seven days, you may only want to redistribute shares every five minutes or so. That provides over 2000 redistributions within a half-life, which is intuitively sufficent for good convergence.

### 2.2.2.3 Node Hierarchy Configuration

Given the two fundamental functional blocks share node and share redistribution node we can now define how the hierarchical FSS tree is constructed. The general configuration is for a share node/ redistribution node pair to be embedded within the tree. These nodes are connected in the fashion shown in Figure 2-9.

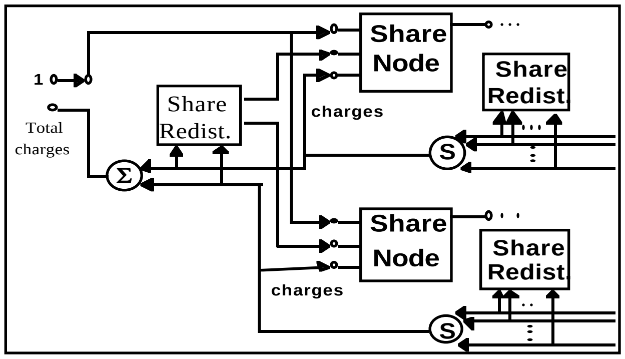


Figure 2-10: Root Node Configuration.

There are also two special cases for node configuration. One case is for children of the root node, and the other is for terminal nodes. For children of the root node, the share redistribution node still exists, but the need for an "advocate" share node is no longer necessary. The diagram for the root node configuration is shown in Figure 2-10. The base_share parameter for children of the root node is always 1.

Finally, there is the terminal node configuration shown in Figure 2-11. It is just the opposite of the root node configuration in that the terminal nodes still require an "advocate" but not a share redistribution. The share node terminator, illustrated in Figure 2-12, is fed by the base_share of the terminal node to convert the share information into a CPU scheduler Pri (pri increment).

In this case, a value of $\text{nprocs/share}^2$ is used because it provides a closer approximation to the exponential properties of the decaying pri scheduler. Since the feedback control of the terminal share node will adjust for error, the square is not necessary, but it provides faster convergence.

### 2.2.2.4 Administration Issues

In order to have a properly functioning hierarchical FSS of the type proposed here, there are several key parameters that must be set appropriately. The low-pass filter parameter, $K_C$,

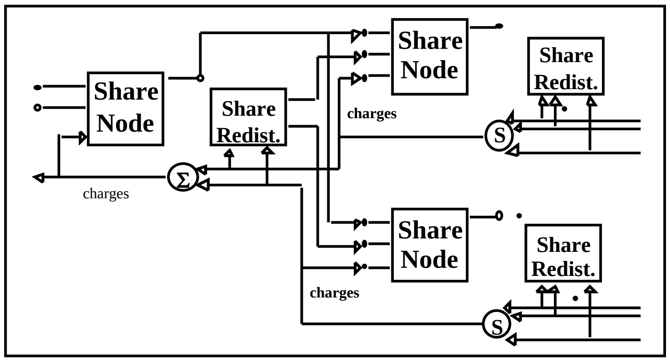


Figure 2-9: Embedded Share.

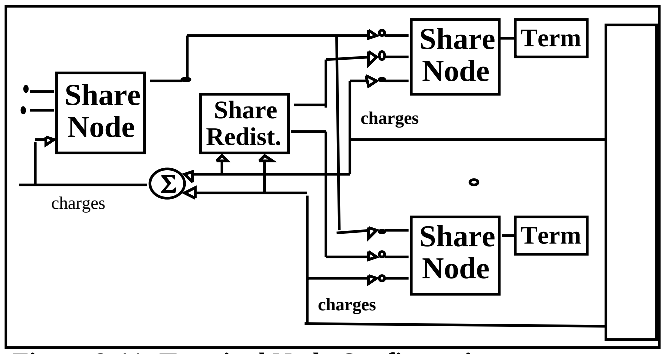


Figure 2-11: Terminal Node Configuration.

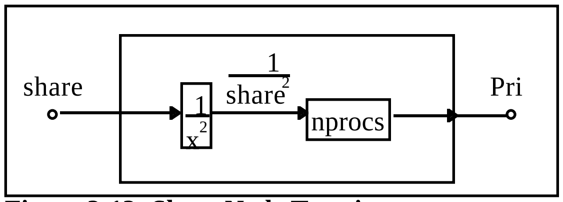


**Figure 2-12; Share Node Terminator.**

must be set to a window of time that is suitable for error-block input. The error-block input must also have appropriate $K_\mu$ values that are commensurate with desire for fast convergence as well as an increase in the parameter based on node depth to guard against thrashing.

The concept of a single usage decay half-life also disappears with the hierarchical implementation. The scheduler gives preference to parents over children when it encounters discrepancies in realized utilization versus entitlement, so some mechanism must exist to provide fairness to both. The example discussed in 2.2.1.2 and illustrated by Figure 2-3 clarifies this issue. Given the same scenario as this example, we can step through the mechanism of the proposed hierarchical FSS and see how it responds. First, 4 runs as the lone process for an extended period. Each time the share redistribution runs, organization A receives a greater share to compensate for its lack of utilization, and within organization B, 3 receives a greater share. Since 4 is still the only active user, it still receives all the processing time. Later, when 1 and 2 from organization A log in and begin to run, their organization's share is much greater than organization B's due to share redistribution, so their processes enjoy the majority of the processing time regardless of what is running under organization B. For the purpose of discussion, assume that 4 continues to run and 3 logs in and begins processing. It is now clear that 3's performance will be poor relative to its entitlement, but good relative to 4. Assume 3 continues to run. All nodes are now active and 3 is the only one that is not realizing its entitlement. Eventually the usage of organization B will decay and the usage for organization A will accrue to the point that their redistributed share will equalize. 3 then will enjoy more processor time, but the usage for 4 will also have been decaying this whole time. So by the time the organizations have corrected themselves, 4 will also have done its penance and resume requesting the majority of its entitlements. In this scenario 3 was unable to realize its entitlement. A solution to this problem would be to increase the decay half-life for the children 3 and 4. Then, when the organizations have adjusted to correct their usage discrepancies, 3 will receive the majority of organization B's entitlements until either by decay of 4's usage or increase of 3's, the ratio of their usages equals the ratio of their entitlements.

## 3 Enhancing Fair Share Predictability

With algorithmic improvements in Section 2.2.2, Fully Hierarchical Feedback Approach, FSS becomes a correct entity in and of itself. However, Fair Share Scheduler can still encounter situations where the actual utilization as shown in periodic accounting reports will miss the target utilization's for one or more levels of the share hierarchy. These reasons all stem from the use of Fair Share Scheduler as the principal means of throughput management and reveal themselves in the failure to saturate the share hierarchy nodes. This section addresses those issues.

The analysis of Fair Share Scheduler led to a classification of the issues into three categories:

- Appropriate Utilization (Node Saturation)
- Resource Manager Interoperability
- Tuning Incompatibilities with Fair Share Paradigm

### *3.1 Appropriate Utilization*

The foremost reason for failure for the realized utilization to match the expected entitlements closely is lack of node saturation. Briefly stated, it means that if a user doesn't run any jobs, he doesn't accumulate usage. However, it extends beyond the no job case to include the case where the user does not run enough work at any one time to drive the CPU utilization of the system to his entitlement percentage. Node saturation refers to the CPU workload executing for a given *lnode*. An lnode is the data structure which exists in the kernel for a resource group or shareholder and which contains all information required for FSS to function. A saturated lnode is one where sufficient runnable processes are present in the system to utilize the entitlement defined for that resource group or shareholder.

Fair Share Scheduler operates only on runnable processes when there is contention for the CPUs. Idle time suspends FSS scheduling, and each scheduler interval with idle time contributes to the cumulative deviation from the target utilization (entitlement).

### *3.2 Interference with Non-Fair Share- Controlled Resource Managers*

Fair Share Scheduler has limited interaction with other system resource managers. This allows other significant resource constraints to become predominate in determining throughput. An improved extension of the Fair Share Scheduler would exchange information with the other resource managers on the system. Alternatively, all resource management could be gathered together into a single module to provide an integrated global resource management.

#### *3.2.1 Memory Scheduling*

Processes are not runnable when they are swapped out, waiting on I/O devices or other resource allocations. All of these resources are controlled by managers which do not consider system usage history when making allocation decisions. Memory allocation management (swapping) is one of the largest factors controlling a process's ability to connect to a CPU. Swap decisions are typically based on the size of the memory requirement, the residency time in memory or on the swap device, and the base nice value of the process. However, there is no mechanism to utilize fair share information when trying to determine if a process should be swapped in or out. This can lead to a significant discrepancy between a process's entitlement and its realized utilization. For example, a large memory process executed by a user with a large share entitlement (a common occurrence in many HPC environments) can be swapped out of memory for considerably longer durations and achieve a lower overall utilization than a small memory process executed by a user with a lower entitlement.

This requirement for interaction between Fair Share Scheduler and the UNICOS memory manager has been previously recognized as a requisite. A means of implementing that critical link was developed for a Fair Share Scheduler implementation supporting a strictly timed job cycle co-resident with ongoing, ad hoc development jobs. This method was deemed successful and is described in [2].

### *3.2.2 I/O Scheduling*

I/O requests to disk, tape, and network devices are typically scheduled in a FIFO fashion. No means of considering fair share information exists. This can give rise to situations where processes are repeatedly unable to realize their full effective share. When considered over a longer time interval, such as an accounting interval of a week or month, repeated occurrences of this interference will result in a subnominal realized (actual) CPU utilization.

Two cases illustrate this phenomenon. In the first example, two processes from two different groups having equal entitlements are in execution. Both perform equivalent levels of I/O activity. Process A originates from a user with a high recent usage history while, Process B originates from a user with a low recent usage history. Fair Share Scheduler will lower the effective share of Process A and raise the effective share of Process B. This will allow Process B to gain more CPU time than Process A so that the two users' utilization will converge on their entitlements. However, each process generates roughly the same number of I/O requests, and each I/O request will have the same probability of being scheduled during any given time. Under the typical case where a process must complete the I/O request before continuing execution, they both experience the same amount of I/O wait time. This will interfere with Process B achieving its effective share, and the impact becomes more pronounced as the I/O boundedness of these processes increases. If the Fair Share information was utilized, I/O requests for Process B could be scheduled ahead of Process A I/O requests such that they will have a probability of being serviced equivalent to the effective share of Process B.

In the second case, two processes from two different groups having different entitlements are in execution. Process A performs sporadic, light I/O and originates from a user with a low recent usage history. Process B performs steady, heavy I/O and originates from a user with a high recent usage history. Both processes operate in the typical case where I/O must be completed before continuing execution. In this case, Process A's occasional I/O requests will be starved out behind Process B's steady stream of requests, and Process A will spend a disproportionate time waiting on I/O. Fair share has acted to boost Process A's effective share to compensate for the low recent usage history, and yet the runability of Process A suffers behind the continual stream of I/O generated by Process B. Using Fair Share information, Process A's requests could be scheduled sooner than most of Process B's requests so that it can resume execution and operate closer to the effective share set for it by Fair Share.

The second case has implications beyond traditional process executions. I/O operations result in calls to library routines and cannot be performed in parallel regions of a multitasked code. Therefore, the problem becomes compounded by the additional processes which must wait at a semaphore or barrier before continuing. The effect can also be particularly damaging to performance for applications in a DCE. Assume that Process A is an interactive session where Process B is a batch application performing I/O using DFS. If both processes must share the same network at any point, the interactive session response would be adversely impacted.

It has been argued that this effect is insignificant because the processes with lowered effective share would use less CPU time and therefore generate fewer I/O requests. However, it can be easily shown that high I/O request rates can be generated by a very small segment of code requiring a very small amount of CPU time. Further, the size of the I/O data transfer has no relationship to the amount of CPU time required by the process. It has also been argued that when a process has a large entitlement, the usage history decay accrued during the wait interval will cause FSS to allocate more CPU resources to it when the I/O completes. The actual interval is not significant in terms of most practical decay half-lives (milliseconds vs. hours). It is more significant that a waiting process is not accumulating usage and is therefore falling off its effective share.

#### *3.2.2.1 Device Allocations*

Other resource allocations, such as for tape drives, Secondary Data Segments, and scratch disk space, present wait times to a process when there is contention for that device. The wait times degrade the process's ability to realize its required effective share. System actions such as data migrate and recall actions, *ldcache* buffer and system cache management can increase wait times for a process. Actions that cause wait times in a way which does not consider fair share usage information will cause the actual process execution times to vary from that which FSS has computed is necessary to meet the target utilization. In short, FSS considers CPU time alone, does not consider any of the reasons why a process may not be able to accumulate usage, and provides no mechanism for many of these resource managers to consider CPU usage history when arbitrating contention.

## *3.3 Tuning Inconsistencies*

Inconsistencies in the tuning of a system can lead to discrepancies between the entitlements and the accounting reports. These can be either intentional or unintentional.

Intentional inconsistencies arise from tuning goals which implicitly ignore the share entitlements. For example, the stated goal of minimizing total wait time in the job input queues runs or establishing an upper bound on the acceptable job expansion factor [(input queue wait + wall clock run time)/CPU time] run counter to the fair share paradigm. By definition, Fair Share Scheduler is employed to favor certain groups of users over others. This places emphasis on jobs submitted by the favored groups and implies that their jobs should be scheduled more frequently, even to the point of degrading access to the system for the unfavored groups. This will cause jobs from the unfavored groups to languish in the input queue while the system processes work from the favored groups. Adjusting the system to minimize the input queue wait time requires that jobs from the unfavored groups be occasionally elevated in importance above that which is appropriate for their FSS entitlement. Likewise, the expansion factor would require the same adjustments to the input queue, compounded by similar priority boosts to the job once in execution. The net effect can be to raise the realized utilization above the entitlement for the unfavored groups at the expense of realized utilization for the favored groups. While this would be acceptable to the groups with lower entitlements, the groups with higher entitlements will find this objectionable.

Unintentional conflicts are created when inappropriate parameters on other resource managers contradict share entitlements. Typically, this would be caused by an incomplete understanding of the parameter and its full range of side effects to other system resource managers in general and Fair Share Scheduler specifically. Such cases are most often the result of the significant complexity to which UNICOS and its product set has evolved and not through a lack of individual qualifications.

The UNICOS implementation features the ability to utilize usage history in the Unified Resource Manager [4]. This provides a means to order jobs from resource groups with low recent usage relative to their entitlements with higher priority than other jobs. This increases the probability that a high-entitlement job will be gated into execution at the expense of low-entitlement jobs. This is clearly a desirable feature if conformance to the entitlements is a high priority.

Likewise, if tuning goals are expressed as part of the NQS queues structure and certain categories of jobs are favored, this will tend to skew the actual utilization for groups whose jobs fit the categories. The URM ranking algorithm provides a means, via the NQS Interqueue Priority, to rank jobs based on the queue (rmgr: /urm/service_wt).

It is the user's responsibility to saturate his or her node with favored job types. For example, the HPCMP seeks to foster development of multitasking technology. NAVOCEANO MSRC is tuned to favor multitasking jobs at the URM/NQS level. However, this creates potential for deviation from FSS entitlements when actual system utilization is reviewed in the monthly accounting reports. Several groups exist where the predominate workload is multitasked, yet these groups have a small total system entitlement. This tends to gate more of their jobs into execution via the preferred multitasking queues as opposed to groups with higher entitlements that have a predominately unitasked workload and are, therefore, constrained to use queues which are not preferred. A solution to this aspect of the problem is impossible utilizing FSS or FSS enhancements. Users must saturate their nodes by developing multitasking codes. In this case, FSS scheduler provides an implicit method for realizing the DoD's goal of developing multitasking software technology. Groups who make this investment will be more likely to realize their allocated entitlement instead of unintentionally surrendering portions of their share to groups with lower allocations.

### *3.4 Internal FSS Issues*

As was discussed earlier; there are issues relating to the hierarchical implementation that make it prone to unpredictability. Resolving these issues would result in a more predictable system.

Part of the unpredictability of the FSS is the relationship between the scheduler delta and the usage decay half-life. Granularity of the scheduler is partially a function of the ratio of scheduler delta to usage decay half-life. Granularity is reduced as the ratio is reduced, so decreasing the scheduler delta (which may be computationally expensive) or increasing the half-life will provide a higher resolving power. Another limit is the share quantum defined by the *sharemin* (shradmin -Z) and the group error quantum *mingshare* (shradmin -Y).

Edge effects are also a product of very low decayed usage compared to other users currently executing processes. For a short time (depending on the length of the usage decay half-life, possibly a very long time) a user can dominate the system.

## 4 Conclusion

It remains the responsibility of the users to submit sufficient appropriate workload to the system in order to receive their share allocation. The definition of appropriate workloads is the responsibility of the controlling authority of the system. It remains incumbent on the operational management to assure that the technical decisions and implementations, that is, tuning parameters and job queues, etc., are established consistent with this definition. However, as shown in this paper, there are significant issues with system-wide implications in the current implementation of FSS and its interactions with the other UNICOS resource mangers which degrade the ability to construct a system environment which conforms to the controlling authority's requirements.

## 5 Acknowledgments

We would like to thank Mr. Chris Brady, Cray Research, for his patient explanation of the UNICOS Fair Share internals and his role as sounding board for discussions on the algorithmic improvements to FSS. We would also like to thank Dr. Alan Wallcraft, Naval Research Labs, and Ms. Melody Clifford, NAVOCEANO, for their analysis and observations which exposed the discrepancies in FSS behavior. A special thanks goes to Ms. Virginia Sotler, Army Corps of Engineers, Waterways Experiment Station (CEWES). We are extremely appreciative of the pioneering work she performed in establishing the DoD FSS environment and for sharing with us her experiences and operational knowledge, not to mention the source for the actual DoD Share Tree. Finally, are especially grateful for the efforts of Mr. Michael McLendon for help with hashing out mathematical details, and Mrs. Mary Ellen Jones for editorial and logistical support.

## 6 References

1) Brady, C., DOE/Richland Fair Share Scheduler, Internal White Paper

2) Brady, C., Thorp, D., and Pack, J., Priority Based Memory Scheduling, *Proceedings of the Thirty-Third Semi Annual Cray User Group* Meeting, pp 221-223, Spring 1994

3) Cray Research, Inc., *UNICOS System Administration, Volume 2,* SG-2113 8.0, pp 511. 1994

4) Cray Research, Inc., *UNICOS Tuning Guide,* SR-2099 8.0, pp 121-125. 1994

5) DDR&E Testimony to the Subcommittee on Defense Technology, Acquisition, and Industrial Base of the Senate Committee on Armed Services. Subject: Plans for Reducing and Modernizing the RDT&E Infrastructure of the DoD. April 22, 1994. http://www.acq.osd.mil/ddre/docs/4-22-1994_Testimony.html

6) Kay, J., and Lauder, P., A Fair Share Scheduler, *Communications of the ACM*, Vol 31, No. 1, pp 44-55. January 1988.

7) Phillips and Harbor, Feedback Control Systems, 2nd Ed., Prentice Hall, 1988

8) Zinnel, K.C., What Every Administrator Should Know Before Trying to Set Up a Share Hierarchy in the UDB, *Proceedings of the Thirty-Fifth Semi Annual Cray User Group* Meeting, pp 221-223, Spring 1995

***PostScript error (--nostringval--, --nostringval--)***