Opportunity Scheduling:
An Unfair CPU Scheduler for UNICOS

Richard Klamann
Los Alamos National Laboratory

mail stop B269, Los Alamos, NM 87545
voice: (505)665-3181 fax: (505)667-0168
email: rmk@lanl.gov

ABSTRACT: *Fair Share is the standard scheduling algorithm used for political resource control on large, multi-user UNIX systems. Promising equity, Fair Share has instead delivered frustration to its Los Alamos UNICOS users, who perceive misallocations of interactive response within a system of unreasonable complexity.*

This paper reviews the design of the Kay/Lauder Fair Share system, as well as its Cray UNICOS implementation, and concludes that the underlying model is inappropriate for interactive control. A new resource manager, Opportunity Scheduling, is then presented. Salient features of the new scheduler include: (1) direct management of interactivity (or "computing opportunity"), (2) job prioritization within resource groups, (3) cooperative memory scheduling, and (4) a simple, user-oriented interface.

The paper then contrasts Opportunity Scheduling with the batch system employed at Los Alamos, where throughput and cycle allocation, rather than computing opportunity, are paramount considerations. It concludes with anecdotal experiences under the system.

1. Introduction

Perhaps the most direct way to get an operating system up and running on a mainframe supercomputer is to port one designed for a smaller machine, scaling the table sizes and clock rates accordingly. The results, however, will be less than satisfying. Because mainframes are expensive, their cost, and hence rights to the machine, are often split between many different interest groups. And because supercomputers provide coveted performance, their cycles are valuable. This combination yields one new quantity in abundance: competition.

Resource control, or rather the lack of it, is a well-known shortcoming of the UNIX operating system. In the relatively pastoral setting of a departmental workstation, this flaw is a minor one. Users of departmental systems are, by definition, cooperative; they share an interest in the successful completion of the group's work. If the group is small, its members can easily administer its resources, without help from the operating system, by simply talking to one another. Should any user habitually flood the departmental machine with unimportant work, his supervisor can correct the situation, often by means more effective than those available to any operating system.

In the dog-eat-dog world of mainframe computing, where hundreds of individuals from dozens of independent organizations share rights to a single machine, such informal controls are absent; anonymous users have little incentive to cooperate. Work viewed as "unimportant" to one department may be "vital" to another. Upper level management, the only administrative recourse for such disputes, will simply not be bothered.

In this larger environment the operating system must ensure equitable access to those parties that have purchased rights to the machine. To provide this control, Cray Research has incorporated the Fair Share CPU Scheduler into its UNICOS operating system [1].

2. Fair Share Design

For most of its users, the Fair Share Scheduler is associated with the *share tree*, its most visible data structure. *Share trees* are a simple mechanism by which successive layers of management can independently subdivide computing resources (or anything else) amongst their subordinates. This division is done by assigning relative weights, or *shares*, to each node within the tree. These weights are then normalized and propagated down the tree, yielding an absolute allocation for each end user.

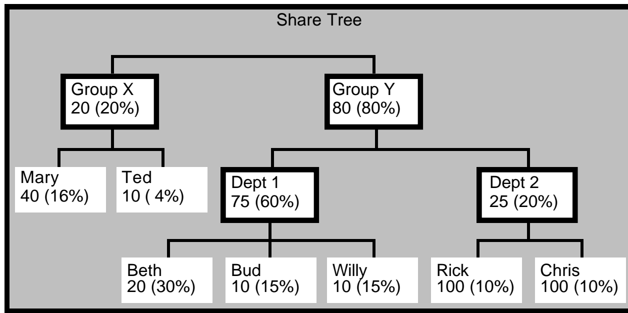

Figure 1: Shares and resulting allocation

Once the share tree is established, Fair Share's mission is to partition CPU cycles *fairly*. Kay and Lauder define a *fair* scheduler as one that allocates resources to users according to each individual's *entitlement*, where *entitlement* is a function of both share and past usage [2]. Under such a scheme, if user *A* has consumed fewer cycles than *B*, but was allotted an equal share of the machine, user *A* is *entitled* to more cycles. To correct this imbalance, a *fair* scheduler will prefer *A's* work to *B's* until *A's* recorded usage has caught up with *B's*.

2.1. Fairness vs. Interactivity

This definition of fairness corresponds to our everyday use of the term, making it both comfortable, and dangerous. The phrase is dangerous because, in its common use, it refers to storable, time-independent quantities. *Fair share* applies comfortably to an inheritance, the size of a piece of a pie, or to the water in a canteen on a long desert trek. We do not speak of a driver, barreling down the interstate at 80 MPH after enduring a wait in traffic, as getting his *fair share* of speed; speed is not a storable, time-independent quantity.

When applied to CPU cycles, *fair share* fosters the illusion that CPU cycles can be stored in a bucket for later consumption. Although users will readily acknowledge the absurdity of this in extreme situations (once cannot obtain a month's worth of computing during a single week), they continue to believe that cycles can be stored for a "reasonable" period of time, typically days, and then delivered to them on demand irrespective of other system activity.

To make matters worse, Fair Share users tend to believe that their interactivity -- the degree to which the system responds to them -- is also being managed fairly. They expect that each individual will receive the degree of interactivity that he deserves, independent of the manner in which others compute. These twin objectives, responsiveness proportional to share, and fair cycle consumption, are usually incompatible.

Consider the simple case of a single processor shared equally by two users, *A* and *B*. The cycles delivered to *A*, who logs on at noon, will depend a great deal upon whether *B* is an early-bird or not. Should *B* computes all morning, *A* will receive scheduling priority for the entire afternoon, or until such time as he has consumed the same number of cycles as *B* -- a half-day's supply. But if *B* waits to compute until noon, *A* and *B* will share the machine equally throughout the remainder of the day, each receiving a quarter-day's supply. Hence the afternoon cycles delivered to *A* depend not only on *A's* usage history, but on *B's* as well. Unless *A* is well acquainted with *B's* habits, *A's* acquisition of afternoon computing is unpredictable.

This unpredictability of cycle delivery can bring about a crisis for true interactive users. Suppose that early-bird *B* has an important demonstration of some new interactive software he is developing scheduled for the afternoon. If *B* computes during the morning, he will saddle himself with horrible system response time for the entire afternoon, or at least until late-arriving *A* is no longer regarded as under-serviced. *B's* optimal strategy, given that interactivity for the afternoon is his prime concern, is to make sure that he is not over-serviced relative to *A*. Under a strictly *fair* scheduler, the only way for *B* to accomplish this is to delay his computing, idling the machine for the entire morning!

2.2 Timely Computing

The primary motivation for purchasing a supercomputer is speed, or *rate* of cycle delivery. Supercomputers customers are not simply interested in consuming cycles; cycles can be gotten more economically from smaller machines. Supercomputers are purchased by users who are primarily interested in the *rate* at which cycles can be acquired.

This is the basic reason why Fair Share misses the mark for interactive users. Fair Share views the aggregate number of consumed cycles, not the rate at which cycles are consumed, as the end quantity to be managed. Instead of providing interactive users with a mechanism for predicting and controlling response time, Fair Share varies the cycle delivery rate in complex and (to users) seemingly unpredictable ways, in order to balance past consumption. Rate of delivery, the supercomputer's *raison d'être*, is made subordinate to accumulated usage.

2.3 Adjustments to Fair Share

To be fair, Fair Share incorporates many features that address the need for the timely delivery of computation. The most basic adjustment, considered integral to the scheduler, is to periodically decay the accumulated usage for each user. By artificially removing imbalances in usage, Fair Share provides some measure of service to everyone, irrespective of past consumption. Early-bird *B*, assuming a rate of decay sufficiently large, can expect some degree of interactivity after late-comer *A* arrives, although the exact amount will depend on the details of *B's* morning usage, other load on the machine, and factors yet to be discussed.

But decay is unclean. Very large decay rates, those that rapidly discard all past usage, contradict the primary design goal of the scheduler -- fairness. He who arrives the first'est with the most'est, wins. And small decay rates, while more or less preserving the scheduler's fair behavior, do little to mitigate its problems with interactive response.

Even with decay, imbalances in usage result in situations that can only be solved by placing usage caps and service minimums in the Fair Share algorithm. Usage caps preserve the normalization step employed by the scheduler at the cost of giving excessive cycles to low priority processes. Service minimums prevent process marooning by ensuring that all jobs, even those belonging to over-serviced users, are given cycles. These kludges both violate fairness, and worse, tend to encourage destructive computing practices, such as task flooding (see section 5).

2.4 Sharing a Surplus

Apart from issues of *fairness*, Fair Share must also allocate to each user his proper *share* of the machine. This is harder than it seems because groups, not individuals, typically own rights to the system, yet consumption is accounted to individuals. Whenever an individual fails to submit sufficient work to consume his share, the cycles delivered to his group will also fall short of its allocation. Should a group have large numbers of relatively dormant user accounts (a common occurrence), the resources allocated to that group will fall well short of its nominal share.

Kay and Lauder address this flaw by assigning shares to organizations, or *resource groups*, and then dynamically subdividing these as their constituent users log in and out. Since logged-in users are not necessarily active users, this technique merely substitutes one problem for another. A group feedback control added to UNICOS attempts to correct this, but with little observed success [3].

Even where proper *share* can be calculated, Fair Share can lose its ability to allocate resources accordingly because it is exclusively a CPU scheduler. With no useful coordination between the memory and CPU schedulers, as memory demand exceeds available core, it is the memory scheduler, and not Fair Share, that determines which processes compute, and which do not.

3. Opportunity Scheduling

The constitution of the United States does not guarantee happiness, only the right to pursue it. The scheduler that we have integrated into the UNICOS operating systems at Los Alamos follows this capitalistic model -- it guarantees the opportunity to compute, not the delivery of computation. Opportunity Scheduling differs from Fair Share in three key respects:

- Past usage (individual or group) has no bearing on present service,
- Resources are allocated to groups, not to individuals, and
- CPU and memory scheduling are coordinated.

To first approximation, Opportunity Scheduling works by simply dividing the physical machine into a set of virtual minicomputers known as *banks*. *Banks* correspond to entities that own rights to the machine, typically departmental organizations. User processes are assigned to, and consume resources exclusively from their department's bank; each bank is assigned a minimum ability to compute commensurate with its rights to the machine. This ability to compute, or computing *opportunity*, is independent of both past usage and demand in other banks.

The scheduler does not, as Fair Share does, further subdivide each bank's allocation amongst the users of that bank. Instead, Opportunity Scheduling uses nice values to prioritize computation within each bank. Users may adjust the nice values of their processes either up or down; the login default is 30; upper and lower bounds are 21 and 38. Each nice decrement results in a 20% increase in service relative to other processes within the bank. Nice adjustments have no influence on the relative performance of processes within a bank relative to those in other banks.

3.1 A Simple Example

With this simple start, let's see how Opportunity Scheduling handles our earlier two-person example. Since resources are allocated to banks, and not to individuals, we must consider two cases:

First, suppose that users *A* and *B* are in separate banks with identical allocations. Under Opportunity Scheduling, each user will be given equal rights to the machine throughout the afternoon. Past usage, for either user, plays no role in scheduling the machine. Should *A* be absent from the machine during the morning, he will not be compensated later; compute resources are not "stored". Neither does an excessive load in any one bank affect the resources given to another; the service accorded to *B's* bank is independent of both the number of processes spawned by *A*, and their nice values.

Now suppose that *A* and *B* compute within a single, machine-wide bank. This situation is more complex. Again, past usage plays no part in scheduling the machine, however the composition of the current workload within the bank, both in terms of number of processes and their nice values, will determine the fraction of the machine delivered to each user. If *A* runs two processes and *B* runs one, each process running at nice 30, then *A* will receive two-thirds of the machine, and *B* will receive one-third. If *B* desires half of the machine, he can either run a second process (yielding four active processes within the bank, half of which belong to *B*), or, equivalently, he can lower the nice value of his single process to 26, doubling its scheduling weight.

Note that "compute resources" are cited in the above example, not simply CPU cycles. Under Opportunity Scheduling, memory is allocated in a manner similar to CPU allocation. The recent time-weighted core consumption for each bank is monitored; should swapping become necessary, processes from banks that have exceeded their memory allocations will be swapped to disk. Within such a bank, high-nice (low priority) processes will spend proportionally more time on the swap device than their higher priority siblings.

Since users may spawn as many jobs as they wish within their banks, and may assign nice values at will to their own processes, the workload within a bank is completely unregulated. This is both the greatest strength, and the greatest weakness of Opportunity Scheduling. To the extent that users are cooperative within their banks, this scheme recreates the efficient, flexible, departmental-machine scenario described in the introductory section. To the extent that users within a bank are competitive, anarchy within the bank results. The success of the scheduler depends upon correctly partitioning the system's users into a small number of moderately sized banks, where the users within each bank have an interest in their bank's work as a whole.

3.2 CPU Scheduling

The UNICOS CPU scheduler, with neither Fair Share nor Opportunity Scheduling active, resembles that of other UNIX systems [4]. A small integer CPU priority is associated with each process; the smaller the integer, the better the priority. At each scheduling event (once per second, or at state transitions) the scheduler ensures that those runable processes with the smallest priorities are connected to CPUs, round-robin scheduling processes with equal priorities. The policy implemented by the scheduler (fair, unfair, or whatever) is realized by the algorithm used to calculate process priorities.

3.2.1 Priority Calculation

Under Opportunity Scheduling, recent connection history is stored in a floating point *penalty* value associated with each process. At each *minor cycle* (60 times a second), all processes connected to CPUs at that moment have their penalty values increased by a process-specific penalty rate:

```
for all connected processes
   Penalty[p] += Rate[p];
```

CPU priorities are then recalculated at each *major cycle* (once per second) by mapping the accumulated penalties for all processes into the 0 to 999 range.

Penalty rates depend on current bank activity and nice values, and are recalculated just before their use. They are based largely on process and bank run weights calculated at each major cycle:

```
Wt[p] = NiceWt[p]*state[p];

Wt[b] = sum(Wt[p]) for all p in b;
```

A process with a nice value of 30 is assigned a nice weight of 1.0. Each incremental change in the nice value alters this weight by 20%.

State factors are estimates of the probability that the process will be runable in the near future:

- 1.0 if runable and in-core.
- 0.8 if a soft sleeper (non-interruptable).
- 0.2 if runable and swapped.
- 0.0 if a hard sleeper (waiting for terminal input, death of a child, etc.).

Once the run weight sums are known, the penalty rate for any process within that bank is given by:

```
Rate[p] = Wt[b]/(CPUs[b]*Wt[p]);
```

where the fractional number of CPUs assigned to a bank is derived from the share tree. This definition of rate turns out to be the inverse of the time that the process will be scheduled to a CPU. For example, a process rate of 4.0 results in the scheduler connecting that process to a CPU 25% of the time during the upcoming interval. (Processes with zero run weights -- those that were not expected to be runable -- are given a penalty rate of 1.)

3.2.2 Multi-CPU Banks

On multi-CPU systems, individual banks will frequently be granted allocations larger than a single CPU. Should any process within such a bank receive a process penalty rate of less than 1.0 (implying that it is to receive more than 100% of a CPU, an impossible goal), the scheduler readjusts the penalty calculations, effectively moving this additional priority to other processes within the bank. Rather than making a separate pass and increasing the overhead of the scheduler, this test and corrective recalculation is placed in the minor cycle:

```
if (Rate[p] && Rate[p] < 1.0) {
   Wt[b] -= Wt[p];
   CPUs[b]--;
   Rate[p] = 1.0;
}
```

Heuristically, this code removes both the high-priority process's weight and a single CPU from the bank's allocation, and assigns an entire CPU to the high-priority process. The remaining processes within the bank then share this additional allocation. This can lead to similar levels of service for processes with different nice values running within the same bank.

3.2.3 Penalty Decay

In order to preserve the exact relationship between penalty rates and CPU allocations, accumulated penalties are "decayed" by subtraction, rather than by small number multiplication. At the end of each major clock cycle, the largest penalty among the *connected* processes is compared to a configurable maximum penalty limit, and if greater, the difference is removed from all process penalties during the next major cycle:

```
for all connected processes
   Adj = max(Penalty[p])-Limit;

if Adj > 0
   for all processes
      if (Penalty[p] > Adj)
         Penalty[p] -= Adj;
      else
         Penalty[p] = 0;
```

"Decay" serves mainly to bound the preferential treatment that newly initiated or recently awakened processes receive under the scheduler.

3.3 Memory Scheduling

Cray machines do not employ virtual memory, hence the UNICOS memory scheduler is a swapper: the entire process image is moved between core memory and secondary storage [4]. The algorithm for deciding which processes are to be resident in core, and for what length of time, is complex; parameters include nice value, residency time, process size, fragmentation, process category (hog, non-hog; batch, interactive), kernel locking, shared text, and thrashing constraints. Other than nice, there are no useful factors indicating the relative importance of work, and since nice values are set at the discretion of individuals independent of share, they are, in isolation, a poor indication of job priority.

Opportunity Scheduling employs the UNICOS memory scheduler with one major modification: the nice parameter is replaced by a memory entitlement factor. A large entitlement for a process raises its swap priority, and thus its average core residency time. The entitlement factor combines the relative importance of a process within its bank, with the memory entitlement for that bank.

```
ME[p] = NiceWt[p]*ME[b];
```

Each bank's memory entitlement reflects the memory demand of all processes within the bank relative to its allocation. Absent nice considerations, a bank's entitlement is simply that bank's core allocation divided by the memory demand of all runable processes within the bank. To prevent process nice values from having a scheduling effect outside of their bank, the size of each process is multiplied by its nice weight when calculating the bank's memory demand:

```
ME[b] = SysMem*Alloc[b]/Demand[b];

Demand[b] = sum(NiceWt[p]*Mem[p])
                          for all p in b;
```

This scheme results in smaller memory entitlements for banks that over-subscribe their allocations, and larger entitlements for those banks that live within their means. When factored into swap priorities, it prevents any one bank from crowding out the work of another.

3.4 User Interface

Perhaps the clearest indication that something is wrong with the Fair Share model lies with the number and complexity of the tools used to monitor its behavior. Not counting commands that modify scheduling and share-tree settings, Cray supports half a dozen Fair Share performance monitoring utilities. The most basic utility, SHRVIEW, produces ten distinct report types depending on invocation, and by default dumps a flat table of floating point numbers typically in excess of one hundred lines. Strangely, none of these utilities detail the performance of individual processes, typically the issue of most concern to users.

Under Opportunity Scheduling, the priority given to a process is determined by (1) the relative share allocation for the bank in which the process is running, and (2) the number and nice values of processes running within the bank. Hence there are just two performance monitoring tools provided:

SYSVIEW shows the current share allocations and usage for every active bank on the machine. Its purpose is to assist managers in setting bank allocations, and to monitor system load. In practice SYSVIEW is employed by users to get an overview of scheduling on the system, to gain confidence in the scheduler, and to lobby management for larger allocations.

```
SYSVIEW: rho    10 second sample   (Nov 27 16:05:16)

                   Processors          Core Memory
BANK    SHARE    ALLO  REQ  USE     ALLO   REQ   USE
----    -----    ----  ---  ---     ----   ---   ---
Root              8.0 11.6  8.0    115.5 125.7 105.8
  PROD  <10>      0.8  2.0  0.8     11.6  72.3  52.4
  INT   <85>      6.8  9.6  7.1     98.2  39.7  39.7
    CIC   <10>    0.7  1.0  0.9      9.8   4.2   4.2
    X     <20>    1.4  5.1  2.2     19.6  15.9  15.9
    DNA   <25>    1.7  1.0  1.0     24.6   4.6   4.6
    POOL  <45>    3.1  2.5  2.5     44.2  14.9  14.9
      T     <50>  1.5  0.0  0.0     22.1   4.5   4.5
      OTH   <50>  1.5  2.5  2.5     22.1  10.4  10.4
 *SYS   < 5>      0.4  0.0  0.0      5.8   2.1   2.1
```

Figure 2: SYSVIEW output for *rho*, a YMP-8/128

BANKVIEW displays summary information for the given bank, along with a detailed breakdown of all active processes within the bank. Process level information includes nice value, memory residence, and CPU utilization. By isolating and displaying the load within a single bank, BANKVIEW exposes any competition that may be present. It is then up to the bank members, through informal, administrative means, to ensure that the number of running jobs and their respective nice values are appropriate for the bank.

```
BANKVIEW: rho    4 second sample   (Nov 27 16:15:19)

              ======================
              | Resource Bank: X    |
              ======================
      ALLO  REQ  USE              ALLO    REQ    USE
  CPU:  1.4  5.0  2.3      MW:   19.6   13.1   13.1

   User   PID   TTY @ CPU  NI Minutes Mwords  Command
   ----   ---   --- - ---  -- ------- ------  -------
roderic 40479    no R  35% 34   112.6    3.4  vm2
    rep 44362 p053 R  42% 34    58.6    1.3  lahetx0
    rep 45319 p053 R  25% 34    43.9    1.6  mcnp
  kammj 45608 p075 R  25% 34    38.9    4.2  xskru.1
   mcnp 57136 p018 C 100% 30     0.4    1.0  mcnp
    rep 26489 p053 S   0% 30     0.1    0.1  csh
```

Figure 3: BANKVIEW output for X-Division's bank on *rho*

3.5 Redistribution of Share

Because Opportunity Scheduling allocates resources directly to banks, rather than to individuals, idle users will not skew the allocations given to groups. Even so, banks will frequently have idle resources -- CPU and memory allocations which exceed their current requirements. Figure 2 (above) illustrates this situation. Consider for the moment CPU allocations: banks T and SYS have no active processes, yielding 1.5 and 0.4 idle CPUs respectively, and the single processes running under DNA cannot consume more than one of the 1.7 CPUs allocated to that bank. Under Opportunity Scheduling, these surplus CPUs will be redistributed to other banks with no future repayment obligation. The question is, which banks should benefit from this windfall?

In the original implementation, idle resources were redistributed amongst all leaf banks (those with processes attached) in proportion to their share allocations, without regard for the hierarchical relationships implied by the share tree. Using this flat scheme, and assuming the situation above, surplus cycles from bank T would be distributed throughout the tree, with banks OTH and X benefiting similarly (having nearly equal CPU allocations).

The flat method was abandoned because it did not take into account the political relationships implied by the share tree. Management expected surplus cycles (as well as memory) to be reallocated to nearest relatives: first to siblings, then to uncles and cousins. Using a hierarchical scheme, and again referencing figure 2, bank T's surplus cycles should first be given to OTH, and only if OTH is unable to consume them should the excess be distributed further.

Hierarchical redistribution of CPUs is accomplished by annotating the share tree with the current processor demand, and then sorting the tree each cycle by relative demand (number of active processes divided by CPU allocation). A breadth-first pass is then made through the tree, recovering surplus CPUs from the first nodes visited and reassigning them to later siblings in proportion to share. This procedure is repeated for core memory. (An early version of Fair Share described in [5] uses hierarchical run queues to allocate CPUs equivalently.)

4. Production Computing

Although the focus of Opportunity Scheduling is on interactive computing, 70% of the supercomputing cycles at Los Alamos are consumed by production -- batch work submitted to and controlled by PROD, a locally developed batch control system. PROD consists of a central batch scheduler, which runs on a dedicated workstation, and a set of batch servers, one per Cray, which regulate the production load for each machine. Users submit jobs, adjust parameters, and receive status information for current, pending, and recently completed work by running a simple network interface to the central scheduler.

Production is the preferred way to run lengthy, unattended work. At night and on weekends PROD schedules the minimum number of jobs necessary to fully utilize the hardware; once a job is selected for execution, it runs at near real time. This strategy minimizes system overhead, turnaround time, and loss from crashes. PROD jobs also survive scheduled downtime via the UNICOS checkpoint and restart mechanism. Reflecting these efficiencies, production jobs are charged 85% of the interactive rate, and are given off-shift scheduling priority.

Opportunity Scheduling partitions the machine between interactive and production work by placing user processes into one of two special high-level banks: interactive sessions run within departmental banks subordinate to "INT"; production jobs are attached to "PROD". The fraction of the system given to each type of work is determined by share assignments to these special banks. A third bank, "SYS", controls system overhead; daemons, cron jobs, and login sessions for system administrators are attached to this bank.

The production scheduler resembles Opportunity Scheduling in its treatment of groups and their work. Each job submitted to the PROD system is placed in a production queue corresponding to the bank of the user that submitted it. The bank members establish ahead of time a job selection algorithm for their queue based on departmental policies, including such factors as user-assigned priority, time limit, and time in queue. Using this algorithm, each queue chooses a candidate job. The scheduler then calculates a service ratio for each queue and master queue, based on the queue's recent usage, allocation, and time limit of its chosen job. Whenever a production slot opens up (i.e., the batch server for a worker machine determines that there is room for another job), the production scheduler runs the chosen job from the least serviced queue belonging to the least serviced master queue.

The PROD system resembles Fair Share, and differs from Opportunity Scheduling, in that it factors the decayed usage for each bank into its job selection algorithm. PROD attempts to balance the usage for each bank against that bank's share of the machine, where usage includes both interactive and batch computing, PROD monitors both. Usage is decayed over 2-3 days. (Longer decay times cause the turn-around time for production work to be unacceptably dependent on the work habits of users in other banks.)

PROD differs from Fair Share in that it balances consumption between banks (both past and projected) strictly by job selection, not by varying the rate of cycle delivery. Fair Share's goal of continuously balancing delivered cycles against user share will often maximize both job turnaround time, and exposure to system failure.

5. Experiences

We were somewhat surprised by system performance under Opportunity Scheduling. No scheduler can create cycles out of thin air, the best it can do is to allocate resources to the right processes. So we were not disappointed when, at its introduction on a saturated

machine, Opportunity Scheduling seemed to have no impact on the overall performance of the system. Yet after several days, the responsiveness of the machines improved markedly. We now understand why.

Fair Share sets an incentive trap for its users. New users are given excellent response when entering a Fair Share system because they are new, and thus have no recent usage history. Only once they are significantly into a computing project and have built up some usage will the load on the machine become apparent to them. Loath to discard the time already spent, users will tend to stay with a project that they might otherwise never have initiated. Meanwhile, more newcomers are enticed onto the system, slowing response even more.

To make matters worse, frustrated users, stuck on the system anyway, tend to submit even more work. Surprisingly this helps, not hurts, the submitting user: Users who run multiple jobs build up large usage histories quickly, which, by design, dampen their ability to acquire more cycles. However, since decay is multiplicative, these larger usages are more quickly decayed. This decay boost ensures that multiple jobs run simultaneously will, collectively, acquire a larger share of the machine than if they were run sequentially. While advantageous to the individual user, when this practice is replicated, the additional load brings the machine to its knees.

These incentive traps do not exist under Opportunity Scheduling. Since usage history plays no part in scheduling, users see the response they can expect right from the beginning; they are less likely to initiate work that cannot be completed in a timely fashion. Since users consume resources exclusively from within their own banks, the full impact of any additional load is immediately felt by, and isolated to, departmental coworkers. Peer pressure from team members (or, lacking that, the action of an immediate supervisor) provides powerful motivation for each individual to compute within the means available to his bank, and to use the production system for non-interactive work.

6. Summary

Opportunity Scheduling differs from Fair Share in that it treats computing as an opportunity, allocable to banks, rather than as a right, allocable to individuals. Users consume resources exclusively from their own bank, and are in direct competition only with fellow bank members. Neither an individual's past usage, nor that of his (or any other) bank, has any bearing on present level of service. Within banks, processes are prioritized by nice value, which their owners may adjust either up or down. Bank share and intra-bank process nice values alone determine both CPU and memory scheduling priorities.

By discarding fairness and its requisite fiction, storable cycles, Opportunity Scheduling is able to directly manage the essence of interactive computing -- rate of cycle delivery. By grouping users that share common rights to a machine and allowing direct work prioritization within these groups, Opportunity Scheduling gives management more flexible control over its computing resource. And by removing incentive traps, Opportunity Scheduling encourages productive computing practices, improving the overall performance of the systems under its control.

7. Implementation Notes and Acknowledgments

Opportunity Scheduling first made its appearance in October 1995, and has been running all of LANL's production PVP Cray hardware since April 1996. At this writing the scheduler is running on two 8-processor YMPs, a J-932, and a T-94. The modified kernel is capable of using either Fair Share or Opportunity, switchable via a runtime flag.

Thanks to Chris Brady of Cray Research, who visited us in April 1995, prior to the conception of Opportunity Scheduling, to explain the inner workings of Fair Share. The memory scheduler mod is based on his work, and we highly recommend his swapper simulation code to anyone needing to manipulate memory scheduling (`nschedv`) parameters.

Thanks to Manuel Vigil and Ray Miller who authorized the manpower for the development effort; to Tom Klingner for the insight given in many discussions; to Rick Light and Don Olivas for tool building; to the entire systems team for defining design requirements; and to Steve Finn and the DNA (now DSWA) user community for agreeing to be the first guinea pigs.

8. References

1. "Fair-share Scheduler", *UNICOS System Administration,* SG-2113, Cray Research Inc.
2. J. Kay and P. Lauder, "A Fair Share Scheduler", *Comm ACM,* Vol 31 No. 1 (Jan. 1988) pp. 44-56
3. C. Everitt and T. Jones, "The UNICOS Fair Share Scheduler as a Feedback Control System", CUG 1995 Fall *Proceedings*
4. J. Bach, *The Design of the UNIX Operating System,* Prentice-Hall, Englewood Cliffs, N.J., 1986
5. Henry, "The Fair Share Scheduler", *Bell Systems Tech. J.,* Vol 63 No. 8 (Oct. 1984) pp. 1845-1857

(Last update: 4/30/97)