Near-Dedicated Scheduling

Chris Brady, CRI, Boulder, Colorado, USA, *Mary Ann Ciuffini*, NCAR, Boulder, Colorado, USA, *Bryan Hardy*, CRI, Boulder, Colorado, USA

ABSTRACT: *With the advent of high-performance workstations, supercomputers are moving from general-purpose scientific computing to handling "Grand Challenge" problems too large for workstations. At NCAR, a mechanism has been developed to provide "Near-Dedicated Scheduling" for expeditious execution of these Grand Challenge problems under UNICOS. This paper presents the problems, issues, and solutions encountered in implementing an integrated near-dedicated scheduler, including: CPU scheduling, memory preemption and SDS preemption. Performance comparisons of near-dedicated vs. dedicated execution in controlled and production environments are also discussed.*

1 Introduction

Scientists computing at the National Center for Atmospheric Research (NCAR) on a saturated CRAY Y-MP 8/64 were getting poor turnaround for their efficient multitasked climate models. NCAR's Scientific Computing Division (SCD) management and technical staff decided that the best way to improve turnaround for these jobs was to offer dedicated computing time slots on the Y-MP 8/64. Thus, the "Dedicated Queue" was created.

From 01:00 to 06:00, if there was at least two hours worth of dedicated work, all Network Queuing System (NQS) batch queues would be stopped and all running jobs would be checkpointed to allow dedicated batch jobs to execute one at a time. Prior to the dedicated start time, the datasets that were to be used by the dedicated jobs were moved from the file server to a directory on the Y-MP 8/64 that was protected from file purging. Each dedicated job had to be multitasked well enough to sustain 88% CPU utilization during its two-hour execution window.

This scheme had several limitations. The major concern was that valuable CPU cycles were being wasted. Even the most efficient dedicated jobs could not use all of the available CPU time, and some of the jobs could not sustain the 88% CPU utilization required for execution in the dedicated queue. In addition, CPU time was wasted during job transitions. The checkpoint and release of normal batch jobs at the beginning and end of the dedicated time slot was expensive and time consuming. Checkpointing these jobs could take up to an hour. Jobs that could not be checkpointed had to be killed, resulting in lost work. The protected directory in which dedicated jobs staged their large datasets prior to run time caused chronic file-system space shortages. As other Cray resources became available at NCAR, users requested that the dedicated time periods be expanded. Due to limitations inherent in the dedicated queue, rather than expanding its window, SCD technical staff and Cray Research, Inc. (CRI) analysts opted to develop a more flexible and efficient near-dedicated (ND) scheme to replace the dedicated queue.

2 Near-Dedicated Scheduling Design

Recognizing that a "start over from scratch" design was the most appropriate way to proceed, a list of features and functionality to strive for in this new implementation was assembled:

- High system utilization -- The primary motivation in pursuing this project was to reclaim unused CPU resources that are normally wasted during dedicated processing. Providing a mechanism for other processes to utilize these idle resources, to "fill in the cracks" without adversely affecting the ND job is imperative to attaining this goal.
- Integrated scheduling -- Three distinct system resources will require scheduling in an integrated, preemptive manner: CPU, memory, and Secondary Data Segment (SDS) space.
- No user level changes -- Other than specifying a different queue to which to submit their job, users should not have to make any code changes or job deck changes in order to run using ND.
- Guaranteed job completion -- A specific time window will exist in which near-dedicated runs are allowed. Because jobs

will not be checkpointed for recovery during the next window, a user's job must not be initiated unless there will be sufficient time for it to complete.

- Flexibility -- Operators must be able to adjust the start-time and/or end-time of the ND window to accommodate system maintenance and other scheduled events that might conflict with ND. Some interaction with individual running jobs may also be appropriate.
- No staging of files required -- Users should not need staging. The previous implementation required that users stage their files by running separate jobs to move all needed files to local disk prior to the dedicated period. This was quite awkward at both the user and administrative levels, and often led to poor disk utilization.
- Performance criterion -- The same site policies should be retained that require users to demonstrate, through **ja** output, a specified level of performance/parallelism in order to be eligible for ND processing.
- Throughput -- The same amount of work should be able to be accomplished during an equivalent time window as was possible using the previous dedicated implementation.

From these criteria, the design for ND was developed and resulted in a set of: NQS and kernel mods, C programs, shell scripts, and cron jobs. Working together, these provide the best of both worlds -- pseudo dedicated processing is easily available to users, while otherwise unused system resources are transparently reclaimed for other jobs. ND does not incur the expensive and time-consuming startup and shutdown costs inherent with dedicated computing. Consequently, no minimum amount of ND work is required as a prerequisite to activate an ND window. Since normal jobs can use the CPUs and memory while an ND job waits to acquire datasets from the file server, prior acquisition of data files is not necessary. Thus, a protected directory is no longer needed. This solves the chronic file-system space problems. Without the need for staging of datasets, ND processing can be started at any time and executed for any length of time.

The following sections will describe many of the components and tools that comprise ND; some of the problems encountered with their solutions; and finally, results from both the testing phase, and after nine months of daily production.

3 Implementation and Pitfalls

The implementation of ND consisted of two separate tasks: creating an integrated priority scheduler, and developing programs and scripts for the operation and monitoring of ND.

3.1 Integrated Priority Scheduling

UNIX, and consequently UNICOS, have limited mechanisms for scheduling processes based on fixed priorities. Nice values are used to reduce the priority of processes, but have limited effect. UNICOS “real-time” scheduling does provide fixed priorities, but because of its absolute nature, it has a number of pitfalls and negative side effects that make it inappropriate for anything but specialized, well-controlled applications. In addition, neither nice nor real-time are able to schedule or preempt the use of SDS. For ND, an integrated scheduler capable of scheduling all system resources (including: CPUs, memory and SDS) was needed. Priorities needed to be adjustable with a much wider range than is currently available by using nice values, but not the absolute behavior of real time.

3.1.1 Establishing Fixed Priorities

To assign fixed priorities to processes, advantage was taken of the existing fair-share scheduler mechanisms in UNICOS. The fair-share scheduler in UNICOS uses kernel lnode structures to establish process priorities. The priority of each process is established by the lnode that the process is associated with. The association of processes to lnodes is, by default, according to user ID (UID). This scheme provides a flexible method for assigning priorities to processes. However, the priorities in fair-share are not fixed and therefore, not very predictable. A simplification of fair-share priority calculation is:

pri = (shares / decayed_usage) / runnable_processes

To provide fixed priorities, a small change was added to the kernel that adds a new fair-share flag (FIXEDPRI) that can be set via the **shradmin** command. When this flag is set the priority calculation essentially becomes:

pri = shares

This provides a fixed priority value that can be easily adjusted and displayed using standard fair-share tools.

Next, these fixed priorities needed to be assigned to various processes. For this implementation of ND, there are three classifications of processes: 1) interactive, 2) normal batch, and 3) near dedicated. To set priorities for each of these process classes, the share structure in Figure 1 was created. This was done by adding “shareholder” entries in the User Data Base (UDB). The PRI_INTR lnode sets the priority for all interactive processes, the PRI_BATCH lnode establishes the priority for all “normal” batch jobs, and the PRI_ND lnode is used to give only the ND jobs high priority. Associating each class of processes to one of these three lnodes creates a fixed priority for each process class. To associate the interactive processes with the PRI_INTR lnode, a standard but little known fair-share flag (DEFERTORESGRP) was employed. For batch jobs, a mod was made to NQS that, based on the name of the queue, associates the job with either the PRI_BATCH or PRI_ND lnode as follows:

```
if (queue_name == near_dedicated_ queue)
    lnode = PRI_ND
else
    lnode = PRI_BATCH
```

With fair-share, system processes are automatically associated with the root lnode, giving these processes the highest priority.

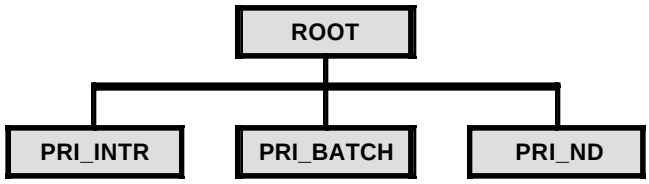

Figure 1. Share Structure for ND

3.1.2 CPU Scheduling

Scheduling CPU resources based on these newly established priorities did not require any changes, since the fair-share mechanisms had been altered to create the priorities. The fair-share mechanisms provided the kind of CPU scheduling behavior that was desired: a wide range of priorities without the absolute behavior of real-time. By using a standard fair-share scheduling flag (NOSCHED), CPU scheduling by priority can be easily enabled or disabled at any time.

3.1.3 Memory Scheduling

Scheduling memory based on fixed priorities proved to be challenging. The UNICOS memory scheduler is quite robust and does a fairly good job of maximizing system throughput. However, there was little support for prioritizing processes, and there are a number of situations where scheduling priorities were being ignored. To implement memory scheduling by priority, there were four problems that had to be solved: 1) coordination of swap and fair-share priorities, 2) hard sleep processes, 3) locked I/O, and 4) swap thrashing

3.1.3.1 Swap and Fair-Share Priorities

The first task was to change the memory scheduler (sched) to use fair-share priorities in the calculation of swap priorities. This was done by reusing the **nschedv** pfactor_in and pfactor_out (-p and -P) parameters that are currently not useful and not recommended for use. The calculation of swap priorities is rather complex and can include many factors. Thus, the discussion here will be restricted to how the portion of the swap priorities controlled by the pfactor parameters are computed. To compute the portion of swap priority controlled by the pfactors, the share priority is normalized and then multiplied by the corresponding pfactor as follows:

in-memory processes:

$$pri\_in = (share\_pri / max\_share\_pri) * pfactor\_in$$

swapped processes:

$$pri\_out = (min\_share\_pri / share\_pri) * pfactor\_out$$

With this change, swap priorities can be controlled by fair-share priorities via the **nschedv** pfactors.

3.1.3.2 Hard Sleep Processes

A serious flaw in sched is termed as the “hard sleep problem.” In sched, there are two categories of sleeping processes -- soft sleep and hard sleep. The idea is that soft sleep processes are waiting for an event that is predicted to complete soon, whereas a hard sleeper is expected to sleep for a much longer period of time. In the interest of keeping runnable processes in memory, if sched needs to free up memory, it will unconditionally swap out a hard sleeper regardless of its swap priority. In most cases, this works well. However, there are situations where very large processes will briefly become hard sleepers and end up swapping in and out continuously. It should also be apparent that this scheme greatly interferes with scheduling based on fixed priorities. The problem was not that hard sleepers are treated differently, but that there was no way to control how they are handled.

To solve this problem, a second value to the **nschedv** tfactor_in (-T) parameter was added that provides control over how hard sleepers are handled. The first value is the time factor for runnable and soft sleep processes, and the second new value is used for hard sleepers. When a process becomes a hard sleeper, the in-core priority is decreased by the new parameter (tfactor_in_hs) value multiplied by the time in seconds that the process has been asleep as follows:

$$pri\_in = old\_pri\_in - (tfactor\_in\_hs * sleep\_time)$$

When the process wakes up, the priority calculation returns to normal and the hard sleep interval will have no effect on priority. This scheme allows the amount of time that a process can hard sleep before being swapped to be controlled, and most importantly this time is relative to the original swap priority.

3.1.3.3 Locked I/O

Another problem with scheduling of memory by priority is locked I/O. Processes doing locked I/O are not considered for swap out regardless of their priority. In most cases, sched can catch these processes later when they are not doing locked I/O and swap them out. However, on occasion there are processes that constantly do locked I/O, making it impossible for them to be preempted from memory. To address this problem, a change was made to sched so that processes doing locked I/O are always considered for swap out. If, after a process has been selected for swap out, sched finds that it can not swap the process due to locked I/O, it marks the process for swap out by setting a new process flag LCKSWP. With this flag set, the process is put to sleep if it attempts to issue additional I/Os. When the outstanding I/Os complete, the process can then be swapped. Sched was also modified to swap processes marked for swap out (via the LCKSWP flag) that no longer have outstanding locked I/O before all other processes. This provided predictable, reliable preemption of processes doing locked I/O.

3.1.3.4 Swap Thrashing

Once the above problems were resolved, an additional phenomenon was observed that the authors call “swap thrashing.” This problem was often seen when a number of

processes needed to be swapped out to make room for a large ND process. For example, given the situation where there are ten processes that need to be swapped, suppose the first nine are swapped without any difficulty. However; if for any reason the tenth process can not be immediately swapped out, then sched gives up on swapping in the large ND process. Sched then looks for other processes that it can swap in, and in most cases. swaps in the nine processes that it had just swapped out. At this point, the whole process starts over again and continues until all ten processes can be immediately swapped out. This severe swap-out thrashing can go on indefinitely. By using the **nschedv** cpu_factor and max_outage parameters, sched can be set up to only consider for swap in the process with the highest priority. This eliminates the thrashing problem, but creates some rather severe utilization and response problems. A compromise was employed that made sched more persistent about bringing in the highest-priority process, but would eventually allow it to give up. A counter was added to sched that is incremented each time a process is either swapped out or marked for swap out. Each pass through the scheduler causes the count to be decremented, and sched is only allowed to give up when the count goes to zero. In this way, the amount of persistence is proportional to the amount of work incurred in trying to bring in a particular process.

3.1.4 SDS Scheduling

UNICOS uses the Unified Resource Manager (URM) to manage the use of SDS, but lacks the ability to schedule SDS based on priority. Since SDS allocations are generally small in number and mostly static, a simple approach was taken for scheduling SDS. A daemon was created that monitors SDS usage. This daemon watches for processes that are waiting on an SDS allocation request. If a process waiting for SDS is found, the priority of the waiting process is compared with the priority of each of the processes with SDS allocations. If there is enough SDS in use by lower-priority processes to satisfy the new request, the lower-priority processes are suspended. The suspend algorithms were designed to suspend the smallest possible number of processes to accommodate the new request. Suspending processes causes their SDS allocations to be released after being written to the swap device, allowing the new request to be satisfied. Once the new request has been satisfied, the suspended processes are resumed. Since these processes need an SDS allocation, they will simply sleep until the SDS is released. The kernel SDS packing code is relied upon to manage fragmentation of SDS space.

3.2 Operational Overview

Two independent mechanisms control scheduling during the ND period. The first is a cron job initiated at regular intervals to reorder the ND input queues such that jobs run in an order determined by site policy. This cron job runs both outside the ND time-window so that users may see the current queue order, and inside the time window so that the queue order will reflect jobs submitted after ND starts.

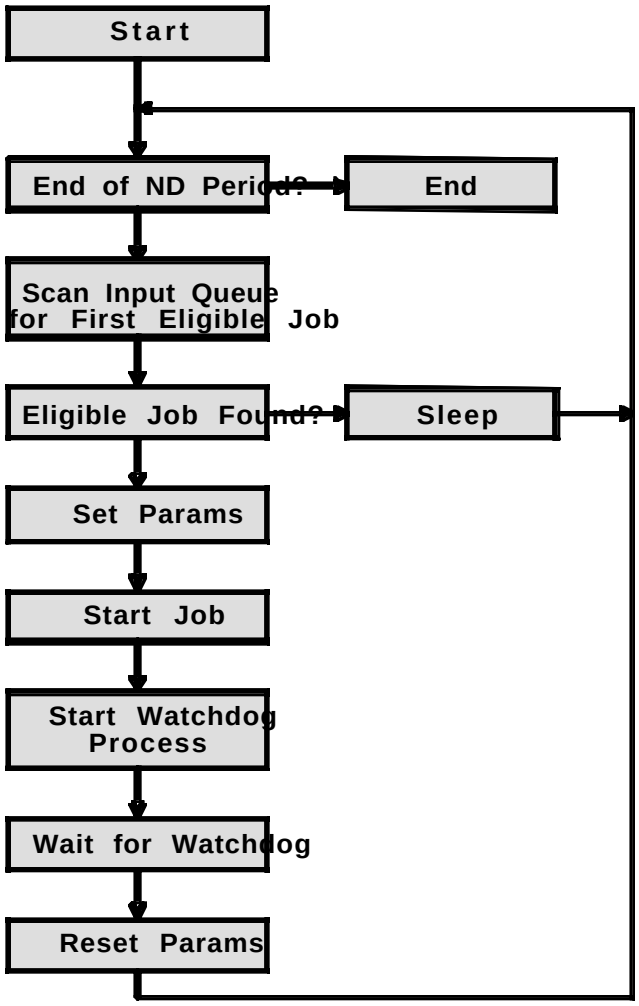

Figure 2. Near-Dedicated Flow

The second control mechanism is the main event loop process. This is started from a manual or cron-initiated script to begin the ND time window. It sets priorities and limits such that the ND control processes will not themselves be restricted by an ND job, and performs other initialization activities. When the event loop is started, it will run for the entire duration of the ND period. The first major activity is to scan the input queues sequentially for the first job that is estimated to complete prior to the end of the ND period. (Because of the "guaranteed completion" criteria, it is undesirable to schedule a job that would not complete before the end of the ND window.) This wall-clock estimate is based on the user's requested CPU time and calculated using the formula:

*est_time = (requested_time * 2) /*

number_of_cpus_on_system

This estimate is generous to allow for tape mounts, I/O wait time, and other valid activities that might cause the job to accrue wall-clock time but not user CPU time.

Prior to initiating the job, preemptive scheduling is turned on. (This is not kept enabled during the ND window unless there is an active job, because the goal is to mirror normal operations as closely as possible during these periods.) This involves invoking **shradmin** and **nschedv** to enable preemptive CPU and memory scheduling, respectively, and initiating the SDS daemon. The ND control process then schedules the job and goes to sleep. A watchdog process is spawned when the job initiates. The estimated completion time calculated earlier is used by the watchdog process to kill the ND job if it does not complete within this estimate.

Upon completion of the job (whether it ended normally, aborted, or was killed by watchdog due to time limits), the watchdog process ends, awaking the ND control process. Preemptive scheduling parameters are reset to normal values, a check is made to see if the ND time-window has ended, and the process repeats. Note that it is sufficient to check for the end of the ND period only between jobs. Because of the time estimate used before scheduling jobs and the presence of a watchdog process to make sure they don't exceed this, there is no need to check the end time while a job is running.

This completion estimate did raise a concern with several users. Due to the generosity in its computation, towards the end of the ND time window it became increasingly hard to find jobs that would fit. Also, some users would rather have their job partially run in the current ND window rather than be delayed until the next window when it could run to completion. For this reason, an NQS attribute, ND_TMC, was added to ND which specifies the user wants to "Take my chances." If this attribute is present, and the job is next in line for scheduling, ND will schedule the job even if its time estimate is too long. The watchdog process will ensure that the job gets killed prior to the end of the time window.

3.2.1 Reporting

A report mechanism was developed to allow easy tracking of ND utilization, overall ND efficiency, and individual job efficiency (see Figure 3).

3.2.2 Monitoring

A single graphical interface was designed to consolidate operator monitoring and control of ND (see Figure 4). It includes the ability to manually start or stop ND if needed, to monitor the current job including its efficiency, to manually kill the job if it's running poorly, and to extend the time window of a job if appropriate.

```
================================================================================
Near Dedicated Performance Report                        Mon Feb 19 23:00:35 1996
================================================================================

Job Summary:
     User    Start    Wall   Setup   Eff  Sbrk  Swap  SwpTm  MxMem  MxSDS  ExSt
   --------  -----  -------  -----  ----  ----  ----  -----  -----  -----  ----
   sbert     23:00  2:12:42   6:52  89.9     0     1     29   27.3  210.0
   sbert     01:13  2:14:50   7:20  88.8     0     3     65   27.3  210.0
   sbert     03:28  2:16:31  12:06  91.2     0     1     18   27.3  210.0

   Average          2:14:41   8:46  90.0     0     1     37   27.3  210.0

System Utilization:
   User    96
   System   4
   Idle     1

              System Usage                          Job Profile

   Time    Usr  Sys  Idl  Mflp      User     Eff   Mem     SDS   Sbrk  Swaps
  -----    ---  ---  ---  ----    --------  ----  -----  ------  ----  -----
  23:05     92    4    4   702    sbert     -1.0    0.4       0     0      0
  23:10     83    5   12   755    sbert     53.6   27.3  430080     0      1
  23:15     96    4    0  1257    sbert     89.9   27.3  430080     0      1
  23:20     96    4    0  1249    sbert     89.9   27.3  430080     0      1
  23:25     97    3    0  1070    sbert     89.7   27.3  430080     0      1
  23:30     96    4    0  1266    sbert     91.1   27.3  430080     0      1
  23:35     97    3    0  1257    sbert     91.0   27.3  430080     0      1
  23:40     96    4    0  1222    sbert     88.3   27.3  430080     0      1
  23:45     96    4    0  1255    sbert     89.7   27.3  430080     0      1
```

Figure 3. Sample ND Report

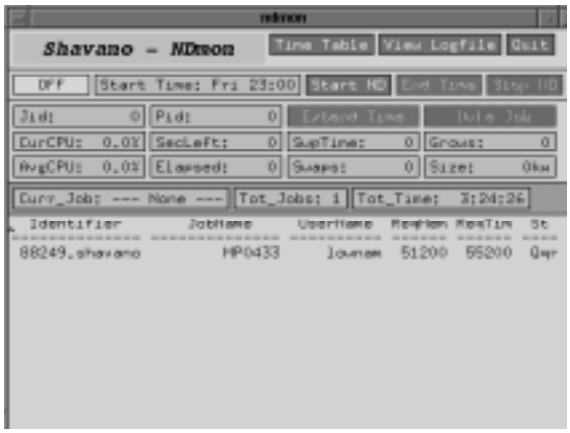

Figure 4. ND Monitoring Tool

3.3 Testing

The existing dedicated queue had long been in production and was used by a number of NCAR's most influential and vocal users. This made effective testing imperative, as disruptions to the current production workload could not be tolerated. Effective testing of ND required a significant amount of dedicated system time. Fortunately, NCAR owns a small CRAY EL92 system that could be used for dedicated testing. This system proved to be invaluable in accomplishing testing goals without impacting production work.

Testing centered around two quite different objectives. The first objective was to verify that the new scheduler changes would deliver quick, reliable preemption of system resources. The scheduler was tested by running a mix of background jobs and then initiating an ND job. The background mix of jobs could be tuned to provide both typical and worst-case workloads. This testing uncovered a number of unexpected problems with memory scheduling, as noted earlier. The second objective was to test the scripts and programs that manage and monitor the operation of ND for reliable operation. This was accomplished by setting up the EL92 for full ND operation and running multiple sequences of jobs under a variety of circumstances.

4 Results

One of the most important objectives for ND was job turnaround. Meeting this objective depended on the performance of the new scheduling mechanisms. Performance in this case is how quickly and completely the new scheduler is able to preempt system resources.

The chart in Figure 5 shows the scheduler's performance. The measurements in this chart were done by running a sequence of real and synthetic jobs in the current dedicated queue, then running the same sequence of jobs using the ND mechanisms

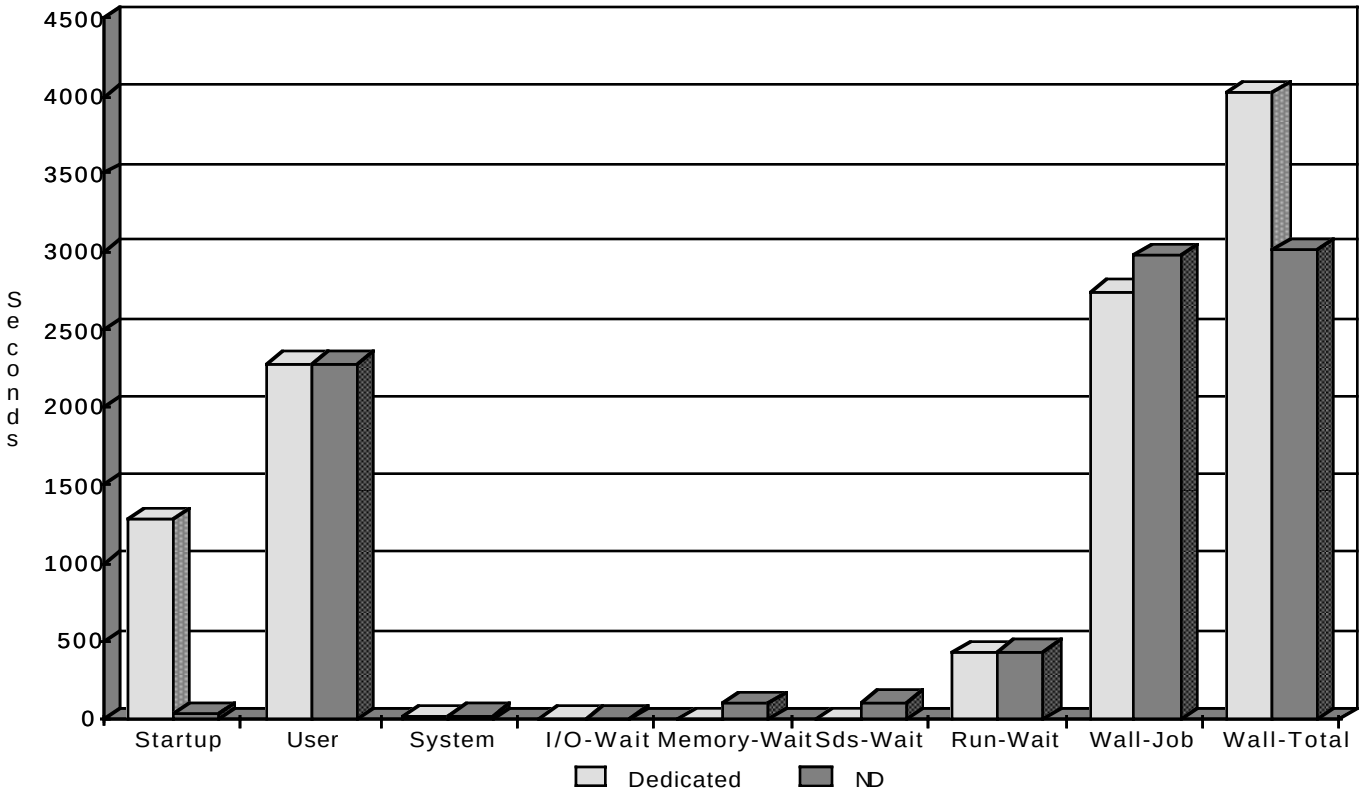

Figure 5. Dedicated vs. Near-Dedicated Execution Times

with a mix of typical background jobs. The startup column is the setup time before the first job is initiated. Since the current workload must be checkpointed before staring the first job in the dedicated queue, this time is quite long. Note that the user, system, and I/O wait times are almost identical with small increases in memory and SDS wait times. The Run-Wait column is derived from the difference in the Wall-Job time and the sum of all other components, and represents the time waiting for CPUs. The Wall-Total column is the start-to-end time for the entire job mix and shows that ND was able execute the job mix, in less overall time than dedicated. In this test case, the average wall time for each job was only 24 minutes. With longer-running jobs, the memory and SDS wait times will be less significant, since these delays only occur once at the start of each job.

The average times for memory and SDS preemption were 22 and 47 seconds, respectively. These were higher than desired, but still acceptable. Preemption of CPU resources was very good. The difference in wall time due to CPU scheduling was only 42 seconds per hour or 1.17%.

High system utilization was also an important objective for ND. Figure 6 shows a comparison of system utilization for dedicated vs. ND. This comparison is based on a 30 day sample. Idle time was reduced from 9.65% to 2.38%, a reduction of 7.27%. The maximum system utilization occurs during ND processing, since during normal processing, idle time for this system is typically 3-6%.

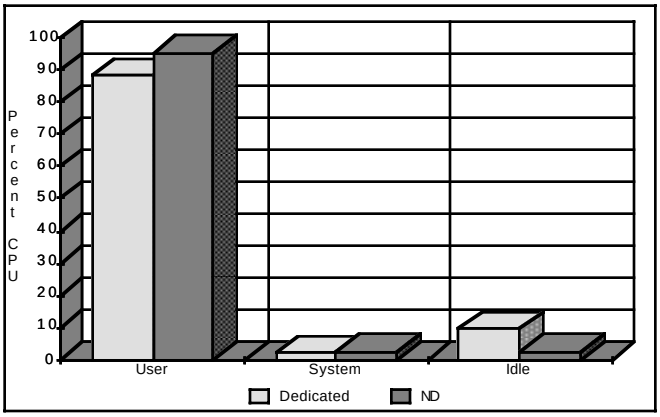

Figure 6. System Utilization

5 Conclusions

Although there were a number of difficult problems to resolve, ND has been very successful. With the exception of a few startup glitches, the scheduling mechanism has been reliable. System utilization during ND processing is typically higher than during normal processing. ND's flexibility has allowed NCAR to expand the hours for ND processing. Many favorable comments from NCAR management and ND users regarding the performance and functionality of ND have been received.