System Performance Tuning

This chapter describes the basics of tuning the IRIX operating system for the best possible performance for your particular needs. Information provided includes the following topics:

- General information on system tuning and kernel parameters. See "Theory of System Performance Tuning" on page 181.
- Tuning applications under development. See "Application Tuning" on page 184.
- Observing the operating system to determine if it should be tuned. See "Monitoring the Operating System" on page 189.
- Tuning and reconfiguring the operating system. See "Tuning The Operating System" on page 205.

Theory of System Performance Tuning

The standard IRIX System configuration is designed for a broad range of uses, and adjusts itself to operate efficiently under all but the most unusual and extreme conditions. The operating system controls the execution of programs in memory and the movement of programs from disk to memory and back to disk.

The basic method of system tuning is as follows:

1. monitor system performance using various utilities,
2. adjust specific values (for example, the maximum number of processes),
3. reboot the system if necessary, and
4. test the performance of the new system to see if it is improved.

Note that performance tuning cannot expand the capabilities of a system beyond its hardware capacity. You may need to add hardware, in particular another disk or additional memory, to improve performance.

Files Used for Kernel Tuning

Table 11-1 lists the files/directories used for tuning and reconfiguring a system.

Table 11-1 Files and Directories Used for Tuning

File/Directory:	Purpose:
*/var/sysgen/system/**	Directory containing files defining software modules
/var/sysgen/master.d	Directory containing files defining kernel switches and parameters
*/var/sysgen/mtune/**	Directory containing files defining more tunable parameters
/var/sysgen/stune	File defining default parameter values.
*/var/sysgen/boot/**	Directory of object files
/unix	File containing kernel image

Typically you tune a parameter in one of the files located in the *mtune* directory (for example, the *kernel* file) by using the *systune*(1M) command.

Overview of Kernel Tunable Parameters

Tunable parameters control characteristics of processes, files, and system activity. They set various table sizes and system thresholds to handle the expected system load. If certain system structures are too large, they waste memory space that would otherwise be used for other processes and can increase system overhead due to lengthy table searches. If they are set too low, they can cause excessive I/O, process aborts, or even a system crash, depending on the particular parameter.

This section briefly introduces some of the tunable parameters and switches. Appendix A, "IRIX Kernel Tunable Parameters," describes all parameters, gives default values, provides suggestions on when to change each parameter, and describes problems you may encounter.

The Types of Parameters

Tunable parameters are specified in separate configuration files in the */var/sysgen/mtune* and the */var/sysgen/master.d* directories. See the *mtune*(4) reference page for *mtune* information and the *master*(4) reference page for information on *master.d*.

The default values for the tunable parameters are usually acceptable for most configurations for a single-user workstation environment. However, if you have a lot of memory or your environment has special needs, you may want to adjust the size of a parameter to meet those needs. A few of the parameters you may want to adjust are listed below.

nproc defines the maximum number of processes, system-wide. This parameter is typically auto-configured.

maxup defines the maximum number of processes per UID.

rlimit-core-cur the maximum size of a core file.

rlimit-data-cur the maximum amount of data space available to a process.

rlimit-fsize-cur the maximum file size available to a process.

rlimit-nofile-cur the maximum number of file descriptors available to a process.

rlimit-rss-cur the maximum resident set size available to a process.

rlimit-vmem-cur
the maximum amount of mapped memory for a process.

sshmseg specifies the maximum number of attached shared memory segments per process.

Application Tuning

You can often increase system performance by tuning your applications to more closely follow your system's resource limits. If you are concerned about a decrease in your system's performance, you should first check your application software to see if it is making the best use of the operating system. If you are using an application of your own manufacture, there are steps you can take to improve performance. Even if a commercially purchased application is degrading system performance, you can identify the problem and use that information to make any decisions about system tuning or new hardware, or even simply when and how to use the application. The following sections explain how to examine and tune applications. The rest of this chapter assumes that your applications have been tuned as much as possible according to these suggestions.

Checking Application Performance

If your system seems slow, for example, an application runs slowly, first check the application. Poorly designed applications can perpetuate poor system performance. Conversely, an efficiently written application means reduced code size and execution time.

A good utility to use to try to determine the source of the problem is the *timex*(1) utility. *timex* will report how a particular application is using its CPU processing time. The format is:

`timex -s` *program*

which shows *program*'s real (actual elapsed time), user (time process took executing its own code), and sys (time of kernel services for system calls) time. For example:

`timex -s ps -el`

The above command executes the *ps -el* command and then displays that program's time spent as:

```
real 0.95

user 0.08

sys 0.41
```

Tuning an Application

There are many reasons why an application spends a majority of its time in either user or sys space. For our purposes, suspect excessive system calls or poor locality of code.

Typically, you can only tune applications that you are developing. Applications purchased for your system cannot be tuned in this manner, although there is usually a facility to correspond with the application vendor to report poor performance.

If the application is primarily spending its time in user space, the first approach to take is to tune the application to reduce its user time by using the *pixie*(1) and *prof*(1) commands. See the respective reference pages for more information about these commands. To reduce high *user* time, make sure that the program:

- makes only the necessary number of system calls. Use *timex -s* to find out the number of system calls/second the program is making. The key is to try to keep **scall/s** at a minimum. System calls are those like *read*(2), *exec*(2); they are listed in Section 2 of the reference pages.
- uses buffers of at least 4K for *read*(2) and *write*(2) system calls. Or use the standard I/O library routines *fread*(3) and *fwrite*(3), which buffer user data.
- uses shared memory rather than record locking where possible. Record locking checks for a record lock for every read and write to a file. To improve performance, use shared memory and semaphores to control access to common data (see *shmop*(2), *semop*(2), and *usinit*(3P)).
- defines efficient search paths ($PATH variable). Specify the most used directory paths first, and use only the required entries, so that infrequently used directories aren't searched every time.
- eliminates polling loops (see *select*(2)).
- eliminates busy wait (use *sginap*(0)).
- eliminates system errors. Look at */var/adm/SYSLOG*, the system error log, to check for errors that the program generated, and try to eliminate them.

Run *timex* again. If the application still shows a majority of either user or sys time, suspect excessive paging due to poor "locality" of text and data. An application that has locality of code executes instructions in a localized portion of text space by using program loops and subroutines. In this case, try to reduce high user/sys time by making sure that the program:

- groups its subroutines together. If often-used subroutines in a loaded program are mixed with seldom-used routines, the program could require more of the system's memory resources than if the routines were loaded in the order of likely use. This is because the seldom-used routines might be brought into memory as part of a page.
- has a working set that fits within physical memory. This minimizes the amount of paging and swapping the system must perform.
- has correctly ported FORTRAN-to-C code. FORTRAN arrays are structured differently from C arrays; FORTRAN is column major while C is row major. If you don't port the program correctly, the application will have poor data locality.

After you tune your program, run *timex* again. If sys time is still high, tuning the operating system may help reduce this time.

There are a few other things you can do to improve the application's I/O throughput. If you are on a single-user workstation, make sure that the application:

- gains I/O bandwidth by using more than one drive (if applicable). If an application needs to concurrently do I/O on more than one file, try to set things up so that the files are in different file systems, preferably on different drives and ideally on different controllers.
- obtains unfragmented layout of a file. Try to arrange an application so that there is only one file currently being written to the file system where it resides. That is, if you have several files you need to write to a file system, and you have the choice of writing them either one after another or concurrently, you will actually get better space allocation (and consequently better I/O throughput) by writing these files singly, one after another.

If you are on a multi-user server, however, it's hard to control how other applications access the system. Use a large size I/O - 16Kb or more. You may also be able to set up separate file systems for different users. With high sys time output from *timex*, you need to monitor the operating system to determine why this time is high.

Looking At/Reordering an Application

Many applications have routines that are executed over and over again. You can optimize program performance by modifying these heavily used routines in the source code. The following paragraphs describe the tools that will help tune your programs.

Analyzing Program Behavior with prof

Profiling allows you to monitor program behavior during execution and determine the amount of time spent in each of the routines in the program. There are two types of profiling:

- program counter (PC) sampling
- basic block counting

PC sampling is a statistical method that interrupts the program frequently and records the value of the program counter at each interrupt. Basic block counting, on the other hand, is done by using the *pixie*(1) utility to modify the program module by inserting code at the beginning of each basic block (a sequence of instructions containing no branch instructions) that counts the number of times that each block is entered. Both types of profiling are useful. The primary difference is that basic block counting is deterministic and PC sampling is statistical. To do PC sampling, compile the program with the **-p** option. When the resulting program is executed, it will generate output files with the PC sampling information that can then be analyzed using the *prof*(1) utility. Both *prof* and *pixie* are not shipped with the basic IRIX distribution, but are found in the optional IRIX Development Option software distribution.

To do basic block counting, compile the program and then execute *pixie* on it to produce a new binary file that contains the extra instructions to do the counting. When the resulting program is executed, it will produce output files that are then used with *prof* to generate reports of the number of cycles consumed by each basic block. You can then use the output of *prof* to analyze the behavior of the program and optimize the algorithms that consume the majority of the program's time. Refer to the *cc*(1), *f77*(1), *pixie*(1), and *prof*(1) reference pages for more information about the mechanics of profiling.

Reordering a Program with pixie

User program text is demand-loaded a page (currently 4K) at a time. Thus, when a reference is made to an instruction that is not currently in memory and mapped to the user's address space, the encompassing page of instructions is read into memory and then mapped into the user's address space. If often-used subroutines in a loaded program are mixed with seldom-used routines, the program could require more of the system's memory resources than if the routines were loaded in the order of likely use. This is because the seldom-used routines might be brought into memory as part of a page of instructions from another routine.

Tools are available to analyze the execution history of a program and rearrange the program so that the routines are loaded in most-used order (according to the recorded execution history). These tools include *pixie, prof,* and *cc.* By using these tools, you can maximize the cache hit ratio (checked by running *sar -b*) or minimize paging (checked by running *sar -p*), and effectively reduce a program's execution time. The following steps illustrate how to reorganize a program named *fetch*

1. Execute the *pixie* command, which will add profiling code to *fetch*:

   pixie fetch

   This creates an output file, *fetch.pixie* and a file that contains basic block addresses, *fetch.Addrs*.

2. Run *fetch.pixie* (created in the previous step) on a normal set or sets of data. This creates the file named *fetch.Counts*, which contains the basic block counts.

3. Next, create a feedback file that the compiler will pass to the loader. Do this by executing *prof*:

   prof –pixie –feedback fbfile fetch fetch.Addrs fetch.Counts

   This produces a feedback file named *fbfile*.

4. Compile the program with the original flags and options, and add the following two options:

   –feedback fbfile

For more information, see the *prof* and *pixie* reference pages.

What About Commercial Applications?

You cannot usually tune commercially available applications to any great degree. If your monitoring has told you that a commercially purchased application is causing your system to run at unacceptably slow levels, you have a few options:

- You can look for other areas to reduce system overhead and increase speed, such as reducing the system load in other areas to compensate for your application. Options such as batch processing of files and programs when system load levels permit often show a noticeable increase in performance. See "Automating Tasks with at(1), batch(1), and cron(1M)" on page 24.

- You can use the *nice*(1), *renice*(1), *npri*(1) and *runon*(1) utilities to change the priority of other processes to give your application a greater share of CPU time. See "Prioritizing Processes With nice" on page 126 and "Changing the Priority of a Running Process" on page 127.
- You can undertake a general program of system performance enhancement, which can include maximizing operating system i/o through disk striping and increased swap space. See the *IRIX Admin: Disks and Filesystems* guide.
- You can add additional memory, disk space, or even upgrade to a faster CPU.
- You can find another application that performs the same function but that is less intensive on your system. (This is the least preferable option, of course.)

Monitoring the Operating System

Before you make any changes to your kernel parameters, you should know which parameters should be changed and why. Monitoring the functions of the operating system will help you determine if changing parameters will help your performance, or if new hardware is necessary.

Receiving Kernel Messages and Adjusting Table Sizes

In rare instances a table overflows because it isn't large enough to meet the needs of the system. In this case, an error message appears on the console and in */var/adm/SYSLOG*. If the console window is closed or stored, you'll want to check *SYSLOG* periodically.

Some system calls return an error message that can indicate a number of conditions, one of which is that you need to increase the size of a parameter. Table 11-2 lists the error messages and parameters that may need adjustment. These parameters are in */var/sysgen/master.d/kernel*.

Table 11-2 System Call Errors and Related Parameters

Message	System Call	Parameter
`EAGAIN` `No more processes`	*fork*(2)	increase *nproc* or swap space
`ELIBMAX` `linked more shared libraries than limit`	*exec*(2)	increase *shlbmax*
`E2BIG` `Arg list too long`	*shell*(1), *make*(1), *exec*(2)	increase *ncargs*

Be aware that there can be other reasons for the errors in the previous table. For example, EAGAIN may appear because of insufficient virtual memory. In this case, you may need to add more swap space. For other conditions that can cause these messages, see the *Owner's Guide* appendix titled "Error Messages".

Other system calls will fail and return error messages that may indicate IPC (interprocess communication) structures need adjustment. These messages and the parameters to adjust are listed in Appendix A, "IRIX Kernel Tunable Parameters."

Using timex(1), sar(1), and par(1)

Three utilities you can use to monitor system performance are *timex*, *sar*, and *par*. They provide very useful information about what's happening in the system.

The operating system has a number of counters that measure internal system activity. Each time an operation is performed, an associated counter is incremented. You can monitor internal system activity by reading the values of these counters.

The *timex* and *sar* utilities monitor the value of the operating system counters, and thus sample system performance. Both utilities use *sadc*, the *sar* data collector, which collects data from the operating system counters and puts it in a file in binary format. The difference is that *timex* takes a sample over a single span of time, while *sar* takes a sample at specified time intervals. The *sar* program also has options which allow sampling of a specific function such as CPU usage (**-u** option) or paging (**-p** option). In addition, the utilities display the data some what differently.

The *par* utility has the ability to trace system call and scheduling activity. It can be used to trace the activity of a single process, a related group of processes or the system as a whole.

When would you use one utility over the other? If you are running a single application or a couple of programs, use *timex*. If you have a multi-user/multi-processor system, and/or are running many programs, use *sar* or *par*.

As in all performance tuning, be sure to run these utilities at the same time you are running an application or a benchmark, and be concerned only when figures are outside the acceptable limits over a period of time.

Using timex

The *timex* utility is a useful troubleshooting tool when you are running a single application. For example:

timex –s *application*

The **-s** option reports total system activity (not just that due to the application) that occurred during the execution interval of *application*. To redirect *timex* output to a file, (assuming you use the Bourne shell, (*sh*(1)) enter:

timex –s *application* **2> file**

The same command, entered using the C shell, looks like this:

timex –s *application* **> file**

Using sar

The *sar* utility is a useful troubleshooting tool when you're running many programs and processes and/or have a multi-user system such as a server. You can take a sample of the operating system counters over a period of time (for a day, a few days, or a week).

Depending on your needs, you can choose the way in which you wish to examine system activity. You can monitor the system:

- during daily operation
- consecutively with an interval
- before and after an activity under your control
- during the execution of a command

You can set up the system so *sar* will automatically collect system activity data and put it into files for you. Just use the *chkconfig*(1M) command to turn on *sar*'s automatic reporting feature, which generates a *sar -A* listing. A *crontab* entry instructs the system to sample the system counters every 20 minutes during working hours and every hour at other times for the current day (data is kept for the last 7 days). To enable this feature, type:

```
/etc/chkconfig sar on
```

The data that is collected is put in */var/adm/sa* in the form *sann* and *sarnn*, where *nn* is the date of the report (*sarnn* is in ASCII format). You can use the *sar*(1M) command to output the results of system activity.

Using sar Consecutively with a Time Interval

You can use *sar* to generate consecutive reports about the current state of the system. On the command line, specify a time interval and a count. For example:

```
sar –u 5 8
```

This prints information about CPU use eight times at five-second intervals.

Using sar Before and After a User-Controlled Activity

You may find it useful to take a snapshot of the system activity counters before and after running an application (or after running several applications concurrently). To take a snapshot of system activity, instruct *sadc* (the data collector) to dump its output into a file. Then run the application(s) either under normal system load or restricted load, and when you are ready to stop recording, take another snapshot of system activity. Then compare results to see what happened.

The following is an example of commands that will sample the system counters before and after the application:

```
/usr/lib/sa/sadc 1 1 file
```

Run the application(s) or perform any work you want to monitor, then type:

```
/usr/lib/sa/sadc 1 1 file
sar −f file
```

If *file* does not exist, *sadc* will create it. If it does exist, *sadc* will append data to it.

Using sar and timex During the Execution of a Command

Often you want to examine system activity during the execution of a command or set of commands. The aforementioned method will allow you do to this. For example, to examine all system activity while running *nroff*(1), type:

```
/usr/lib/sa/sadc 1 1 sa.out
nroff −mm file.mm > file.out
/usr/lib/sa/sadc 1 1 sa.out
sar −A −f sa.out
```

By using *timex*, you can do the same thing with one line of code:

```
timex −s nroff −mm file.mm > file.out
```

Note that the *timex* also includes the real, user, and system time spent executing the *nroff* request.

There are two minor differences between *timex* and *sar*. The *sar* program has the ability to limit its output (such as, the **-u** option reports only CPU activity), while *timex* always prints the **-A** listing. Also, *sar* works in a variety of ways, as discussed previously, but *timex* only works by executing a command. However, this command can be a shell file.

If you are interested in system activity during the execution of two or more commands running concurrently, put the commands into a shell file and run *timex -s* on the file. For example, suppose the file *nroff.sh* contained the following lines:

```
nroff −mm file1.mm > file1.out &
nroff −mm file2.mm > file2.out &
wait
```

To get a report of all system activity after both of the *nroff* requests (running concurrently) finish, invoke *timex* as follows:

```
timex -s nroff.sh
```

Using par

You can use *par* much as you use *sar*, such as

- during daily operation
- consecutively with an interval
- before and after an activity under your control
- during the execution of a command

There is an extensive reference page for *par*(1). See that page for specifics on usage.

Use *par* instead of *sar* when you want a finer look at a suspect or problem process. Instead of simply telling you how much total time was used while your process was executing, like *timex*, *par* will break down the information so you can get a better idea of what parts of the process are consuming time. In particular, use the following command options:

-isSSdu Check the time used by each system call and the intervening time lag.

-rQQ Check process scheduling, to see if it should be run more or less frequently.

When tracing system calls, *par* prints a report showing all system calls made by the subject processes complete with arguments and return values. In this mode, *par* also reports all signals delivered to the subject processes. In schedule tracing mode, *par* prints a report showing all scheduling events taking place in the system during the measurement period. The report shows each time a process is put on the run queue, started on a processor, descheduled from a processor, including the reason that the process was descheduled. The events include timestamps. You can set up the system so *par* will automatically collect system activity data and put it into files for you.

The *par* utility works by processing the output of *padc*(1). This can be done in two ways: *padc* can be run separately and the output saved in a file to be fed to *par* as a separate operation or *padc* can be invoked by *par* to perform the data collection and reporting in one step.

The *par* utility can provide different types of reports from a given set of *padc* data depending on the reporting options that are specified. This is a reason why it is sometimes desirable to run the data collection as a separate step.

Wrap-up of sar, par, and timex

Now that you have learned when and how to use *par*, *sar,* and *timex*, you can choose one of these utilities to monitor the operating system. Then examine the output and try to determine what's causing performance degradation. Look for numbers that show large fluctuation or change over a sustained period; don't be too concerned if numbers occasionally go beyond the maximum.

The first thing to check is how the system is handling the disk I/O process. After that, check for excessive paging/swapping. Finally look at CPU use and memory allocation.

The sections immediately following assume that the system you are tuning is active (with applications/benchmark executing).

Checking Disk I/O

The system uses disks to store data and transfers data between the disk and memory. This input/output (I/O) process consumes a lot of system resources, so you want the operating system to be as efficient as possible when it performs I/O.

If you are going to run a large application or have a heavy system load, the system will benefit from disk I/O tuning. Run *sar -A* or *timex -s* and look at the **%busy**, **%rcache**, **%wcache**, and **%wio** fields.To see if your disk subsystem needs tuning, check your output of *sar -A* against the figures in the following table. (Note that in the tables that follow, the right column lists the *sar* option that will print only selected output, for example output for disk usage (*sar -d*) or CPU activity (*sar -u*).

Table 11-3 lists *sar* results that indicate an I/O-bound system.

Table 11-3 Indications of an I/O-Bound System

Field	Value	sar Option
%busy (% time disk is busy)	>85%	*sar -d*
%rcache (reads in buffer cache)	low, <85	*sar -b*
%wcache (writes in buffer cache)	low, <60%	*sar -b*
%wio (idle CPU waiting for disk I/O)	dev. system >30 fileserver >80	*sar -u*

Notice that for the %wio figures (indicates the percentage of time the CPU is idle while waiting for disk I/O), there are examples of two types of systems:

- a development system that has users who are running programs such as *make*. In this case, if %wio > 30, check the breakdown of %wio (*sar -u*). By looking at the %wfs (waiting for file system) and %wswp (waiting for swap), you can pinpoint exactly what the system is waiting for.
- an NFS system that is serving NFS clients and is running as a file server. In this case, if %wio > 80, %wfs > 90, the system is disk I/O bound.

There are many other factors to consider when you tune for maximum I/O performance. You may also be able to increase performance by:

- using logical volumes
- using partitions on different disks
- adding hardware (a disk, controller, memory)

Using Logical Volumes to Improve Disk I/O

By using logical volumes, you can:

- grow an existing file system to a larger size without having to disturb the existing file system contents.

- stripe file systems across multiple disks—You may be able to obtain up to 50% improvement in your I/O throughput by creating striped volumes on different disks.

  Striping works best on disks that are on different controllers. Logical volumes give you more space without remaking the first filesystem. Disk striping gives you more space with increased performance potential, but you do run the risk that if you lose one of the disks with striped data, you will lose all the data on the filesystem since the data is interspersed across all the disks.

  Contiguous logical volumes fill up one disk, and then write to the next. Striped logical volumes write to both disks equally, spreading each file across all disks in the volume, so it is impossible to recover from a bad disk if the data is striped, but it is possible if the data is in a contiguous logical volume. For information on creating a striped disk volume, see the *IRIX Admin: Disks and Filesystems* guide.

Using Partitions and Additional Disks to Improve Disk I/O

There are some obvious things you can do to increase your system's throughput, such as limiting the number of programs that can run at peak times, shifting processes to non-peak hours (run batch jobs at night), and shifting processes to another machine. You can also set up partitions on separate disks to redistribute the disk load.

Before continuing with the discussion about partitions, let's look at how a program uses a disk as it executes. Table 11-4 shows various reasons why an application may need to access the disk.

Table 11-4 An Application's Disk Access

Application	Disk Access
execute object code	text and data
uses swap space for data, stack	*/dev/swap*
writes temporary files	*/tmp* and */var/tmp*
reads/writes data files	data files

You can maximize I/O performance by using separate partitions on different disks for some of the aforementioned disk access areas. In effect, you are spreading out the application's disk access routines, which will speed up I/O.

By default, disks are partitioned to allow access in two ways, either:

- three partitions: partitions 0, 1 and 6, or
- one large partition, partition 7 (encompasses the three smaller partitions)

On the system disk, partition 0 is for root, 1 is for swap, and 6 is for */usr*.

For each additional disk, you need to decide if you want a number of partitions or one large one and what file systems (or swap) you want on each disk and partition. It's best to distribute file systems in the disk partitions so that different disks are being accessed concurrently.

The configuration depends on how you use the system, so it helps to look at a few examples.

- Consider a system that typically runs a single graphics application that often reads from a data file. The application is so large that its pages are often swapped out to the swap partition.

  In this case, it might make sense to have the application's data file on a disk separate from the swap area.
- If after configuring the system this way, you find that it doesn't have enough swap space, consider either obtaining more memory, or backing up everything on the second hard disk and creating partitions to contain both a swap area and a data area.
- Changing the size of a partition containing an existing file system may make any data in that file system inaccessible. Always do a complete and current backup (with verification) and document partition information before making a change. If you change the wrong partition, you can change it back, providing you do not run *mkfs* on it or overwrite it. It is recommended that you print a copy of the *prtvtoc* command output after you have customized your disks, so that they may be more easily restored in the event of severe disk damage.

Also, if you have a very large application and have three disks, consider using partitions on the second and third disks for the application's executables (*/bin* and */usr/bin*) and for data files, respectively. Next, consider a system that mostly runs as a "compile-engine."

In this case, it might be best to place the */tmp* directory on a disk separate from the source code being compiled. Make sure that you check and mount the file system prior to creating any files on it. (If this is not feasible, you can instruct the compiler to use a directory on a different disk for temporary files. Just set the TMPDIR environment variable to the new directory for temporary files.) Now, look at a system that mainly runs many programs at the same time and does a lot of swapping.

In this case, it might be best to distribute the swap area in several partitions on different disks.

Adding Disk Hardware to Improve Disk I/O

If improved I/O performance still does not occur after you have tuned as described previously, you may want to consider adding more hardware: disks, controllers, or memory.

If you are going to add more hardware to your system, how do you know which disk/controller to add? You can compare hardware specifications for currently supported disks and controllers by turning to your hardware *Owner's Guide* and looking up the system specifications. By using this information, you can choose the right disk/controller to suit your particular needs.

By balancing the most active file systems across controllers/disks, you can speed up disk access.

Another way to reduce the number of reads and writes that go out to the disk is to add more memory. This will reduce swapping and paging.

Checking for Excessive Paging and Swapping

The CPU can only reference data and execute code if the data or code are in the main memory (RAM). Because the CPU executes multiple processes, there may not be enough memory for all the processes. If you have very large programs, they may require more memory than is physically present in the system. So, processes are brought into memory in pages; if there's not enough memory, the operating system frees memory by writing pages temporarily to a secondary memory area, the swap area, on a disk.

IRIX overcommits real memory, loading and starting many more processes than can fit at one time into the available memory. Each process is given its own virtual section of memory, called its address space, which is theoretically large enough to contain the entire process. However, only those pages of the address space that are currently in use are actually kept in memory. These pages are called the working set. As the process needs new pages of data or code to continue running, the needed pages are read into main memory (called "faulting in" pages or "page faults") . If a page has not been used in the recent past, the operating system moves the page out of main memory and into the swap space to make room for new pages being faulted in. Pages written out can be faulted back in later. This process is called “paging” and it should not be confused with the action of “swapping.”

Swapping is when all the pages of an inactive process are removed from memory to make room for pages belonging to active processes. The entire process is written out to the swap area on the disk and its execution effectively stops. When an inactive process becomes active again, its pages must be recovered from disk into memory before it can execute. This is called “swapping in” the process. On a personal workstation, swapping in is the familiar delay for disk activity, after you click on the icon of an inactive application and before its window appears.

When IRIX is multiprocessing a large number of processes, the amount of this swapping and paging activity can dominate the performance of the system. You can use the *sar* command to detect this condition and other tools to deal with it.

Determining whether your system is overloaded with paging and swapping requires some knowledge of a baseline. You will need to use the command under various conditions to determine a baseline for your specific implementation. For example, you could boot your system and run some baseline tests with a very limited number of processes running, and then again during a period of light use, a period of heavy networking activity, and then especially when the load is high and you are experiencing poor performance. The use of your system log book will assist you in making these baseline measurements.

Table 11-5 shows indicators of excessive paging and swapping on a smaller system.

Table 11-5 Indicators of Excessive Swapping/Paging

Important Field	**sar Option**
vflt/s - page faults (valid page not in memory)	*sar -p*
bswot/s (transfers from memory to disk swap area)	*sar -w*
bswin/s (transfers to memory)	*sar -w*
%swpocc (time swap queue is occupied)	*sar -q*

Table 11-5 (continued) Indicators of Excessive Swapping/Paging

Important Field	sar Option
rflt/s (page reference fault)	*sar -t*
freemem (average pages for user processes)	*sar -r*

You can use the following *sar* options to determine if poor system performance is related to swap I/O or to other factors:

-u %wswp Percent of CPU wait time owed to swap input. This measures the time during which every active process was blocked waiting for a page to be read or written. Values of even a few percent indicate a swap I/O problem.

-p vflt/s Frequency with which a process accessed a page that was not in memory. Compare this number between times of good and bad performance. If the onset of poor performance is associated with a sharp increase of *vflt/s*, swap I/O may be a problem even if *%vswp* is low or 0.

-r freemem Unused memory pages. The paging daemon (*vhand*) recovers what it thinks are unused pages and returns them to this pool. When a process needs a fresh page, the page comes from this pool. If the pool is low or empty, IRIX often has to get a page for one process by taking a page from another process, encouraging further page faults.

-p pgswp/s Number of read/write data pages retrieved from the swap disk space per second.

-p pgfil/s Number of read-only code pages retrieved from the disk per second.

If the *%vswp* number is 0 or very low, and *vflt/s* does not increase with the onset of poor performance, the performance problem is not primarily due to swap I/O.

However, when swap I/O may be the cause, there are several possible actions you can take:

- Provide more real memory. This is especially effective in personal workstations, where it is relatively economical to double the available real memory.
- Reduce the demand for memory by running fewer processes. This can be effective when the system load is not interactive, but composed of batch programs or long-running commands. Schedule commands for low-demand hours, using *cron*

and *at*. Experiment to find out whether the total execution time of a set of programs is less when they are run sequentially with low swap I/O, or concurrently with high swap I/O.

- Make the swap input of read-only pages more effective. For example, if pages of dynamic shared objects are loaded from NFS-mounted drives over a slow network, you can make page input faster by moving all or a selection of dynamic shared objects to a local disk.
- Make swap I/O of writeable pages more effective. For example, use *swap*(1M) to spread swap activity across several disks or partitions. For more information on swapping to files and creating new swap areas, see "Swap Space" on page 114.
- If you have changed process or cpu-related kernel parameters (for example, *nproc*), consider restoring them to their former values.
- Reduce page faults. Construct programs with "locality" in mind (see "Tuning an Application" on page 185).
- Consider using shared libraries when constructing applications.
- Reduce resident set size limits with *systune*. See "System Limits Parameters" on page 222 for the names and characteristics of the appropriate parameters.

Checking CPU Activity and Memory Allocation

After looking at disk I/O and paging, next check CPU activity and memory allocation.

Checking The CPU

A CPU can execute only one process at any given instant. If the CPU becomes overloaded, processes have to wait instead of executing. You can't change the speed of the CPU (although you may be able to upgrade to a faster CPU or add CPU boards to your system if your hardware allows it), but you can monitor CPU load and try to distribute the load. Table 11-6 shows the fields to check for indications that a system is CPU bound.

Table 11-6 Indications of a CPU-Bound System

Field	Value	sar Option
%idle (% of time CPU has no work to do)	<5	sar -u
runq-sz (processes in memory waiting for CPU)	>2	sar -q
%runocc (% run queue occupied and processes not executing)	>90	sar -q

You can also use the *top*(1) or *gr_top*(1) commands to display processes having the highest CPU usage. For each process, the output lists the user, process state flags, process ID and group ID, CPU cycles used, processor currently executing the process, process priority, process size (in pages), resident set size (in pages), amount of time used by the process, and the process name. For more information, see the *top*(1) or *gr_top*(1) reference pages.

To increase CPU performance, you'll want to make the following modifications:

- off-load jobs to non-peak times or to another machine, set efficient paths, and tune applications.
- eliminate polling loops (see *select*(2)).
- increase the *slice-size* parameter (the length of a process time slice). For example, change *slice-size* from Hz/30 to Hz/10. However, be aware that this may slow interactive response time.
- upgrade to a faster CPU or add another CPU.

Checking Available Memory

"Checking for Excessive Paging and Swapping" on page 199 described what happens when you don't have enough physical (main) memory for processes.

This section discusses a different problem - what happens when you don't have enough available memory (sometimes called virtual memory), which includes both physical memory and logical swap space.

The IRIX virtual memory subsystem allows programs that are larger than physical memory to execute successfully. It also allows several programs to run even if the combined memory needs of the programs exceed physical memory. It does this by storing the excess data on the swap device(s).

The allocation of swap space is done after program execution has begun. This allows programs with large a virtual address to run as long as the actual amount of virtual memory allocated does not exceed the memory and swap resources of the machine.

Usually it's very evident when you run out of memory, because a message is sent to the console that begins:

Out of logical swap space...

If you see this message:

- the process has exceeded ENOMEM or UMEM.
- there is not enough physical memory for the kernel to hold the required non-pageable data structures.
- there is not enough logical swap space.

You can add virtual swap space to your system at any time. See "Swap Space" on page 114 if you wish to add more swap space. You need to add physical swap space, though, if you see the message:

```
Process killed due to insufficient memory
```

The following system calls will return EAGAIN if there is insufficient available memory: *exec*, *fork*, *brk*, *sbrk* (called by *malloc*), *mpin*, and *plock*. Applications should check the return status and exit gracefully with a useful message.

To check the size (in pages) of a process that is running, execute *ps -el* (you can also use *top*(1)). The SZ:RSS field will show you very large processes.

By checking this field, you can determine the amount of memory the process is using. A good strategy is to run very large processes at less busy times.

Determining the Amount of System Memory

To see the amount of main memory, use the *hinv*(1) command. It displays data about your system's configuration. For example:

```
Main memory size: 64 Mb
```

Maximizing Memory

To increase the amount of virtual memory, increase the amount of real memory and/or swap space. Note that most of the paging/swapping solutions also apply to ways to conserve available memory. These include:

- limiting the number of programs
- using shared libraries
- adding more memory
- decreasing the size of system tables

However, the most dramatic way to increase the amount of virtual memory is to add more swap space. The previous section that described using partitions explained how to do this and it is covered completely in "Swap Space" on page 114.

Tuning The Operating System

The process of tuning your operating system is not difficult, but it should be approached carefully. Make complete notes of your actions in case you need to reverse your changes later on. Understand what you are going to do before you do it, and do not expect miraculous results; IRIX has been engineered to provide the best possible performance under all but the most extreme conditions. Software that provides a great deal of graphics manipulation or data manipulation also carries a great deal of overhead for the system, and can seriously affect the speed of an otherwise robust system. No amount of tuning will change these situations.

Operating System Tuning Steps

To tune a system, you first monitor its performance with various system utilities as described in "Monitoring the Operating System" on page 189. This section describes the steps to take when you are tuning a system.

1. Determine the general area that needs tuning (for example, disk I/O or the CPU) and monitor system performance using utilities like *sar*(1) and *osview*(1M). If you have not already done so, see “Monitoring the Operating System” on page 189.
2. Pinpoint a specific area and monitor performance over a period of time. Look for numbers that show large fluctuation or change over a sustained period; don’t be too concerned if numbers occasionally go beyond the maximum.
3. Modify one value/characteristic at a time (for example, change a parameter, add a controller) to determine its effect. It’s good practice to document any changes in a system notebook.
4. Use the *systune*(1M) command to change parameter values or make the change in the *master.d* directory structure if the variable is not tunable through *systune*. Remake the kernel and reboot if necessary.
5. Remeasure performance and compare the before and after results. Then evaluate the results (is system performance better?) and determine if further change is needed.

Keep in mind that the tuning procedure is more an *art* than a science; you may need to repeat the above steps as necessary to fine tune your system. You may find that you’ll need to do more extensive monitoring and testing to thoroughly fine tune your system.

Finding Current Values of Parameters

Before you can tune your system, you need to know the current values of the tunable parameters. To find the current value of your kernel parameters, use the *systune*(1M) command. This command, entered with no arguments, prints the current values of all tunable parameters on your system. For complete information on this command, see the *systune*(1M) reference page.

Changing Parameters and Reconfiguring the System

After determining the parameter or parameters to adjust, you must change the parameters and you may need to reconfigure the system for the changes to take effect. The *systune*(1M) utility will tell you when you make parameter changes if you must reboot to activate those changes. There are several steps to reconfiguration procedure:

- back up the system
- copy your existing kernel to *unix.save*

- make your changes
- reboot your system, if necessary

Backing Up the System

Before you reconfigure the system by changing kernel parameters, it's a good idea to have a current and complete backup of the system. See the *IRIX Admin: Backup, Security, and Accounting* guide.

Caution: Always back up the entire system before tuning.

Changing a Parameter

After determining the parameter you need to change (for example, you need to increase *nproc* because you have a large number of users) you must first back up the system and the kernel. Give the command:

```
cp /unix /unix.save
```

This command creates a safe copy of your kernel. Through the rest of this example, this is called your old saved kernel. If you make this copy, you can always go back to your old saved kernel if you are not satisfied with the results of your tuning.

Once your backups are complete, you can execute the *systune*(1M) command. Note that you can present new values to *systune* in either hexadecimal or decimal notation. Both values are printed by *systune*.

An invocation of *systune*(1M) to increase *nproc* looks something like this:

```
systune -i

Updates will be made to running system and /unix.install

systune-> nproc

        nproc = 400 (0x190)

systune-> nproc = 500

        nproc = 400 (0x190)

        Do you really want to change nproc to 500 (0x1f4)? (y/n) y

In order for the change in parameter nproc to become effective
/unix.install must be moved to /unix and the system rebooted

systune-> quit
```

Then reboot your system. Also, be sure to document the parameter change you made in your system log book.

Caution: When you issue the *reboot* command, the system overwrites the current kernel (*/unix*) with the kernel you have just created (*/unix.install*). This is why you should always copy the current kernel to a safe place before rebooting.

Creating and Booting a New Kernel With autoconfig

The *systune* command creates a new kernel automatically. However, if you changed parameters without using *systune*, or if you have added new system hardware (such as a new CPU board on a multiprocessor system), you will need to use *autoconfig* to generate a new kernel.

The *autoconfig* command uses some environment variables. These variables are described in detail in the *autoconfig*(1M) reference page. If you have any of the following variables set, you may need to unset them before running *autoconfig*:

- UNIX
- SYSGEN
- BOOTAREA
- SYSTEM
- MASTERD
- STUNEFILE
- MTUNEDIR
- WORKDIR

To build a new kernel after reconfiguring the system, follow these steps:

1. Become the Superuser by giving the command:

   `su`

2. Make a copy of your current kernel with the command:

   `cp /unix /unix.save`

3. Give the command:

   `/etc/autoconfig -f`

   This command creates a new kernel and places it in the file */unix.install*.

4. Reboot your system with the command:

   reboot

Caution: When you issue the *reboot* command, the system overwrites the current kernel (*/unix*) with the kernel you have just created (*/unix.install*). This is why you should always copy the current kernel to a safe place before rebooting.

An autoconfiguration script, found in */etc/rc2.d/S95autoconfig*, runs during system start-up. This script asks you if you would like to build a new kernel under the following conditions:

- A new board has been installed for which no driver exists in the current kernel.
- There have been changes to object files in */var/sysgen/mtune*, master files in */var/sysgen/master.d*, or the system files in */var/sysgen/system*. This is determined by the modification dates on these files and the kernel.

If any of these conditions is true, the system prompts you during startup to reconfigure the operating system:

Automatically reconfigure the operating system? **y**

If you answer **y** to the prompt, the script runs *lboot* and generates */unix.install* with the new image.You can disable the autoconfiguration script by renaming */etc/rc2.d/S95autoconfig* to something else that does not begin with the letter S, for example, */etc/rc2.d/wasS95autoconfig*.

Recovering from an Unbootable Kernel

The following procedure explains how to recover from an unbootable */unix*, and describes how to get a viable version of the software running after an unsuccessful reconfiguration attempt. If you use the *systune*(1M) utility, you should never have to use this information, since *systune* will not allow you to set your parameters to unworkable values.

1. If the system fails to reboot, try to reboot it a few more times. If it still fails, you need to interrupt the boot process and direct the boot PROM to boot from your old saved kernel (*unix.save*).

2. Press the **reset** button.You will see the System Maintenance Menu:

```
System Maintenance Menu
1) Start System.
2) Install System Software.
3) Run Diagnostics.
4) Recover System.
5) Enter Command Monitor.
```

3. Select option 5 to enter the Command Monitor. You see:

```
Command Monitor. Type "exit" to return to the menu.
>>
```

4. Now at the >> prompt, tell the PROM to boot your old saved kernel. The command is:

   `boot unix.save`

   The system will then boot the old saved kernel.

5. Once the system is running, use the following command to move your old saved kernel to the default */unix* name. This method also keeps a copy of your old saved kernel in *unix.save*:

   `cp /unix.save /unix`

Then you can normally boot the system while you investigate the problem with the new kernel. Try to figure out what went wrong. What was changed that stopped the kernel from booting? Review the changes that you made.

- Did you increase/decrease a parameter by a very large amount? If so, make the change less drastic.
- Did you change more than one parameter? If so, make a change to only one parameter at a time.