Sun Performance Tuning Overview

A structured approach to improving application performance based upon insight into the intricacies of SPARC and Solaris.

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Revision -05, December 1992

© 1992 Sun Microsystems, Inc.—Printed in the United States of America.
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19.

The products described in this paper may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS
Sun, Sun Microsystems, the Sun logo, are trademarks or registered trademarks of Sun Microsystems, Inc. UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc. All other product names mentioned herein are the trademarks of their respective owners.

All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCworks, and SPARCompiler are licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK® and Sun™ Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

X Window System is a trademark and product of the Massachusetts Institute of Technology.

Please
Recycle

Contents

Preface ix

1. Structure 11

2. Sourcecode 13

Algorithms 13

Programming Model 15

Compiler And Optimisations 17

Effective Use Of Provided System Functions 18

Linking, Localisation Of Reference And Libraries 19

3. Executable 21

Customising The Execution Environment 21

The Effects Of Underlying Filesystem Type 24

4. Databases & Configurable Systems 27

Examples 27

Hire An Expert! 27

Use Sun's Database Excelerator Product 27

Basic Tuning Ideas 28
5. Kernel 31
Buffer Sizes And Tuning Variables 31
Paging, MMU Algorithms And PMEGS 35
Configuring Out Unused Devices 50
References 50
6. Memory 51
Cache Tutorial 51
Cache line and size effects 52
Cache Miss Cost And Hit Rates For Different Machines 56
I/O Caches 61
Kernel Block Copy 62
7. Windows and Graphics 65
8. Disk 67
The Disk Tuning Performed For SunOS 4.1.1 67
Throughput Of Various Common Disks 67
Sequential Versus Random Access 71
Effect Of Disk Controller, SCSI, SMD, IPI 71
Load Monitoring And Balancing 74
Multiple Disks On One Controller 75
Mirrored Disks 75
9. CPU 79
Architecture and Implementation 79
The Effect Of Register Windows And Different SPARC CPUs . 80

Comparing Instruction Cycle Times On Different SPARC CPUs 82
10. Multiprocessors 89
Basic Multiprocessor Theory 89
Unix On Shared Memory Multiprocessors 92
SPARC Based Multiprocessor Hardware 94
Measuring And Tuning A Multiprocessor 98
Programming A Multiprocessor 100
Deciding How Many CPU's To Configure 101
11. Network 103
The network throughput tuning performed for SunOS 4.1 . . . 103
Different ethernet interfaces and how they compare 103
Using NFS effectively 105
A. References 107

Tables

Table 1	System Broken Down Into Tuning Layers	11
Table 2	Resource Limits	21
Table 3	Trace Output with Comments	22
Table 4	Tuning SunOS Releases	31
Table 5	Default Settings for Kernel Parameters	32
Table 6	Vmstat fields explained	36
Table 7	Sun-4 MMU Characteristics	37
Table 8	SPARC Reference MMU Characteristics	40
Table 9	Application speed changes as hit rate varies with a 25 cycle miss cost	53
Table 10	Virtual Write Through Cache Details	57
Table 11	Virtual Write Back Cache Details	57
Table 12	Physical Write Back Cache Details	58
Table 13	On-Chip Cache Details	60
Table 14	Disk Specifications	69
Table 15	1.3Gb IPI ZBR Disk Zone Map	70
Table 16	Which SPARC IU and FPU does your system have?	83

Table 17 Floating point Cycles per Instruction 86
Table 18 Number of Register Windows and Integer Cycles per Instruction 87
Table 19 MP Bus Characteristics 96
Table 20 Disabling Processors In SunOS 4.1.X 99
Table 21 Ethernet Routing Performance 105

Preface

This paper is an extensive rewrite and update of one originally written for and presented at the Sun User '91 conference of the Sun UK User Group at the NEC, Birmingham, UK. Thanks are due to Daf Tregear and Elwyn Davies of Sun UKUG for imposing the deadlines that forced me to finish what I had started.

The content of this paper is basically a brain dump of everything I have learned over the years about performance tuning. It includes a structured approach to the subject, opinions, heuristics and every reference I could find to the subject.

With competitive pressures and the importance of time to market functionality and bugfixes get priority over designed in performance in many cases. This paper is aimed both at developers who want to design for performance and need to understand Sun's better, and also at end users who have an application and want it to run faster.

There are several good texts on system[1] and network[2] tuning aimed at the system administrator of a network of machines or a timesharing system. This paper takes a different approach and is written primarily from an application developers point of view, although there is much common ground.

1. "System Performance Tuning, Mike Loukides, O'Reilly"

2. "Managing NFS and NIS, Hal Stern, O'Reilly"

The information presented in this paper has all been gleaned from published sources and to my knowledge it does not contain any proprietary information. It is intended to be available for reference by Sun users everywhere.

Since the public release of this paper in September 1991 I have had a lot of feedback from readers. Particular thanks for detailed comments and contributions go to Brian Wong, Keith Bierman, Hal Stern and Dave Rosenthal of Sun and Gordon Irlam of Adelaide University.

All errors, opinions and omissions are mine.

Any comments, corrections or contributions should be made to the address below.

Adrian Cockcroft
Sun Microsystems Ltd.
306 Science Park
Milton Road
Cambridge CB5 4WG
UK

Internet Email: adrian.cockcroft@uk.sun.com
UKnet Email: adrian.cockcroft@sun.co.uk

Structure

There have been many white papers on performance tuning and related topics for Suns. This one attempts to gather together references, provide an overview, bring some information up to date and plug some gaps. A cookbook approach is taken where possible and I have attempted to stick my neck out and provide firm recommendations based upon my own experience. The rate at which the operating system and underlying hardware change is a major challenge to anyone trying to work with computers so older software and hardware is mentioned where appropriate to show how the base level of tuning has improved with time.

The performance characteristics of a system can be viewed in a layered manner with a number of variables affecting the performance of each layer. The layers that I have identified for the purposes of this paper are listed below together with some example variables for each layer. The chapters of this paper look at each layer in turn and the appendix has details of the references.

Table 1 System Broken Down Into Tuning Layers

Layers	**Variables**
Sourcecode	Algorithm, Language, Programming model, Compiler
Executable	Environment, Filesystem Type
Database	Buffer Sizes, Indexing
Kernel	Buffer Sizes, Paging, Tuning, Configuring
Memory	Cache Type, Line Size And Miss Cost
Disk	Driver Algorithms, Disk Type, Load Balance
Windows & Graphics	Window System, Graphics Library, Accelerators
CPU	Processor Implementation
Multiprocessors	Load Balancing, Concurrency, Bus Throughput
Network	Protocol, Hardware, Usage Pattern

Sourcecode

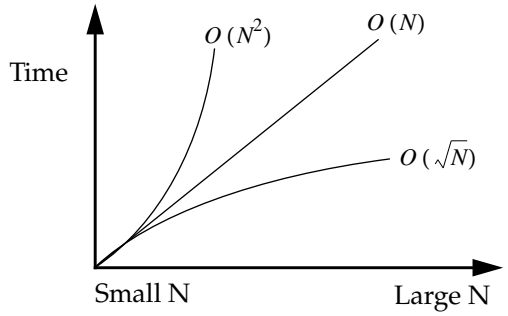
This chapter is concerned with variables that the programmers specifying and writing the software can control.

Algorithms

Many algorithms are invented while writing a program and are as simple to code as possible. More efficient algorithms are often much more complex to implement and may need to be retrofitted to an application that is being tuned. A good book of algorithms[1] and a feel for which parts of the program are likely to be hot spots can make more difference to performance than all other tuning tweaks put together. There are some general classes of algorithms with behaviors as shown below.

Algorithmic Classification

Figure 1 Performance vs. problem size for different classes of algorithms



1. Algorithms by Sedgewick is a digestible alternative to Knuth's definitive reference.

The notation O(N) means that as the amount of data increases by a factor of N the time taken increases by the same order of magnitude. A higher order increase, for example where the time taken increases by the order of N squared, needs to be avoided. Lower order increases are what the textbook algorithms are trying to achieve.

As long as a program is being used with a small data-set the difference between the different classes of algorithms is minor. This is often the case during testing or in the early lifetime of a product.

An Example Problem

One example could be a CAD system that keeps each object in a drawing on a linked list and performs a linear search through the list whenever it needs to find an object. This works for small drawings and it is not too bad if the time taken to perform other operations dominates the execution time. If the CAD system is ported from a Sun3/60 to a SPARCstation 2 then many parts of the code speed up by a factor of perhaps 10 times. The users now expect to run drawings with 10 times the complexity at the same speed as a Sun3/60.

Unfortunately for some operations the time taken to search through the linked list now dominates and the linked list code doesn't see a 10 times speedup due to caching effects (see "A Problem With Linked Lists" on page 54) so there is a performance problem and the algorithm needs to be improved. The solution in this case is to move from a linear search to a more complex search based on hash tables or tree structures. Another approach is to incrementally sort the linked list so that commonly accessed entries are at the beginning.

Space Versus Time

Another issue to consider is space versus time optimization. There's no sense in making an algorithm that's $O(N^2)$ run in O(N) time if doing so requires going from O(N) to $O(N^2)$ space. The increase in storage will probably make the application page, and the disk accesses outweigh any improvement in CPU run time. It is possible to make an algorithm efficient only to use so much space that it doesn't run at the full CPU speed.

Programming Model

There is often a conceptual framework underlying an application which can be thought of in terms of a programming model. Some example models are:

- Hierarchical, structured programming via Jackson Diagrams
- Object Oriented
- Dataflow, the Yourdon/DeMarco method
- State Machines
- Numerical Algorithms
- AI based, rules or knowledge based design
- Forms and records
- Entity relations

Sometimes the model that a programmer or system designer prefers is chosen regardless of the type of system being written. An inappropriate match between the programming model and the problem to be solved is a remarkably common cause of performance problems.

Choice of language to express the algorithm or model

The programmer may decide to choose a language that is familiar to him or may have a language imposed on him. Often the language ends up dictating the programming model, irrespective of the needs of the application, but sometimes languages are forced to implement programming models that are inappropriate. Real examples include a database system written in APL, a real-time control system written in Prolog and a very large message passing object oriented system written in C.

If there is a good match of problem to programming model to language then there is a good chance that the system overall will respond well to tuning. Poorly matched systems sometimes contain so much unnecessary complexity that brute force increases in CPU and I/O throughput are the only thing that has a significant effect.

The moral is that if you come across an application like this and first attempts at tuning have little effect you may as well put your money into buying a faster Sun and your time and effort into redesigning the system from scratch.

It comes down to using the right tool for the job. Most people know what languages to use for particular jobs but some non-obvious points are listed below.

Fortran

Fortran is usually the fastest language for anything that is numerically intensive. For an equivalent level of compiler technology it will always be faster than *C*. This is because it has a simple structure that compilers can handle more easily than C and it doesn't have pointers so there is less chance of side effects during expression evaluation. The key to optimisation is *code motion* and this can be performed more safely in Fortran than in C. A second reason is that Fortran defaults to passing floating point variables by reference (i.e. passing an address rather than the number itself) which is more efficient, especially on processors, like SPARC, that have separate integer and floating point registers and pass parameters in integer registers.

Assembler

In theory assembler is the fastest language possible. In practice programmers get so bogged down in the huge volume of code and the difficulty in debugging it that they tend to implement simple, inefficient algorithms in poorly structured code. It is hard to find out where the hot spots in the system are so the wrong parts of the system get optimised. When you discover a small routine that dominates the execution time, look at the assembler generated by the compiler and tweak the sourcecode. As a last resort consider rewriting it in assembler.

It is often very helpful to understand what sort of code is generated by your compiler. I have found that writing clean simple high level language code can help the compiler to understand the code better and that this can improve the optimisation of the generated assembler. Just think of assembler as read-only code. Read it and understand it but don't try to write it.

C and C++

It seems that just about everything is written in C nowadays. Its biggest weakness is that hardly anyone seems to use lint to check their code as a standard part of their compilation makefiles. Wading through the heaps of complaints that lint produces for most systems written in C gives a pretty good

insight into the sloppy nature of much C coding. Many problems with optimisers breaking C code can be avoided by getting the code to lint cleanly first! ANSI C is an improvement but not a substitute for lint.

C++ should be used whenever an object oriented design is called for. C could be used in most cases but the end result is that sloppy coding and programmers who take shortcuts, combined with the extra level of complexity, make the resulting systems very hard to debug and hence give optimisers a hard time. Writing C++ like code in C makes the code very hard to understand, it is much easier to use a C++ preprocessor, with a debugger and performance analyser that understand the language.

Debug And Test Tools

Lint has just been mentioned, build it into your default makefiles and use it regularly. There is a little known utility called tcov[1] that performs test coverage analysis and produces an annotated listing showing how many times each block of code has been executed and a percentage coverage figure.

There is a product called Purify[2] that can be used to debug subtle errors in C and C++ programs such as used-before-set data, memory allocation errors and array bounds violations. It works by modifying object files to instrument memory references so it can find errors in library routines and it slows down execution by no more than a factor of three.

Compiler And Optimisations

Having chosen a language, there is a choice of compilers for the language. There are typically several compilers for each major language on SPARC. The SPARCompilers from SunPro tend to have the largest user base and the best robustness on large programs. The Apogee C and Fortran compilers seem to have a performance advantage of 10-20% over the commonly used SunPro SPARCompilers 1.0 release for programs (like SPEC benchmarks) that can use maximum optimisation but the Apogee compilers are much less robust. The recent SPARCompilers 2.0 release narrows the performance gap and some preliminary results for SPARCompilers 3.0 show very similar results.

1. See the tcov manual page.

2. Described in a paper presented at Usenix, Winter 92.

Competition between SunPro, Apogee and others will fuel a drive to better compilers. Using compilers effectively to tune code is not covered in this paper since it has been addressed in great depth in previous publications[1][2][3]. To summarise in a sentence: clean C code with lint, turn up the optimiser most of the way, profile to find hot spots and turn up the optimiser for those parts only. Look at the code in the hot spots to see if the algorithm is efficient.

Effective Use Of Provided System Functions

This is a general call to avoid re-inventing the wheel. The SunOS libraries contain many high level functions that have been debugged tuned and documented for you to use. If your system hot-spot turns out to be in a library routine then it may be worth looking at re-coding it in some streamlined way, but the next release of SunOS or the next generation of hardware may obsolete your homegrown version. As an example, the common string handling routines provided in the standard SunOS 4.X C library are simple compiled C code. In Solaris 2 these routines are written in optimised assembler.

There are some very powerful library routines that seem to be under-used.

Mapped files

SunOS 4.0 and all versions of Unix System V Release 4 (SVR4) include a full implementation of the `mmap` system call. This allows a process to map a file directly into its address space without the overhead of copying from kernel to user space. It also allows shared files so that more efficient use of memory is possible and inter-process communication can be performed[4].

Asynchronous I/O

This is an extension to the normal blocking read and write calls to allow the process to issue non-blocking reads and writes[5].

---

1. Numerical Computation Guide, supplied with SunC and SunFortran
2. You and Your Compiler, by Keith Bierman
3. SPARCompiler Optimisation Technology Technical White Paper
4. SunOS 4.1 Performance Tuning
5. aioread, aiowrite manual pages

Memory Allocation Tuning

The standard version of malloc in the Sun supplied libc is optimised for good space utilisation rather than fast execution time. A version of malloc that optimises for speed rather than space uses the 'BSD malloc' algorithm and is provided in /usr/lib/libbsdmalloc.a.

There are some powerful options in the standard version of malloc[1].

- mallocmap() prints a map of the heap to the standard output.
- mallopt() configures quick allocation of small blocks of memory.
- mallinfo() provides statistics on malloc.

The small block allocation system controlled by mallopt is not actually implemented in the code of the default version of malloc or the BSD malloc.

Linking, Localisation Of Reference And Libraries

There are two basic ways to link to a library in SunOS and SVR4, static linking is the traditional method used by other systems and dynamic linking is a run-time link process. With dynamic linking the library exists as a complete unit that is mapped into the address space of every process that uses it. This saves a lot of RAM, particularly with window system libraries at over a megabyte each. It has several implications for performance tuning however.

Each process that uses a shared library shares the physical pages of RAM occupied by the library, but uses a different virtual address mapping. This implies that the library will not be at a fixed point in the address space each time it is used. In fact it can end up in different places from one run of a program to the next and this can cause caching interactions that increase the variance of benchmark results. The library must also be compiled using position independent code which is a little less efficient than normal code and has an indirect table jump to go through for every call that is a little less efficient than a direct call. Static linking is a good thing to use when benchmarking systems but production code should dynamically link, particularly to the system interface library `libc`. A mixture can be used, for example in the following compilation the fortran library is statically linked but `libc` is dynamically linked.

1. malloc manual page

```
% f77 -fast -o fred fred.f -Bstatic -lF77 -lV77 -lm -Bdynamic -lc
```

This dynamically links in the `libc` library and makes everything else static. The order of the arguments is important. This is one of those tricks that is hard to work out and not documented anywhere else so make a note of it! If you are shipping products written in Fortran to customers who do not have Fortran installed on their systems you will need to use this trick.

Tuning Shared Libraries

When using static linking the library is accessed as an archive of separate object files and only the files needed to resolve references in the code are linked in. This means that the position of each object module in memory is hard to control or predict. For dynamic linking the entire library is provided at run time regardless of which routines are needed. In fact, the library is demand paged into memory as it is needed. Since the object modules making up the library are always laid out in memory the same way a library can be tuned when it is built by reordering it so that modules that call each other often are in the same memory page. In this way the working set of the library can be dramatically reduced. The window system libraries for sunview and OpenWindows are tuned in this way since there is a lot of inter-calling between routines in the library. Tools to do this automatically on entire programs or libraries are provided as part of the SPARCworks 2.0 Analyser using some functionality that is only provided in the Solaris 2 debug interface (/proc), linker and object file format. The main difference is that the a.out format used in BSD Unix and SunOS 4 only allows entire object modules to be reordered. The ELF format used in Unix System V.4 and Solaris 2 allows each function and data item in a single object module to be independently relocated.

Executable

This section is concerned with variables that the user running a program on a Sun can control on a program by program basis.

Customising The Execution Environment

Limits

The execution environment is largely controlled by the shell. There is a `limit` command which can be used to constrain a program that is hogging too many resources. The default SunOS 4.1.X resource limits are shown in Table 2.

Table 2 Resource Limits

Resource name	**Default limit**
cputime	unlimited
filesize	unlimited
datasize	524280 Kbytes
stacksize	8192 Kbytes
coredumpsize	unlimited
memoryuse	unlimited
descriptors	64

The most useful changes to the defaults are to prevent core dumps from happening when they aren't wanted:

```
% limit coredumpsize 0
```

To run programs that use vast amounts of stack space:

```
% limit stacksize unlimited
```

To run programs that want to open more than 64 files at a time:

```
% limit descriptors 256
```

Tracing

When tuning or debugging a program it is often useful to know what system calls are being made and what parameters are being passed. This is done by setting a special bit in the process mask via the `trace` command. Trace then prints out the information which is reported by the kernel's system call interface routines. Trace can be used for an entire run or it can be attached to a running process at any time. No special compiler options are required. In Solaris 2 trace has been renamed truss.

Here's some trace output with commentary added to sort the wheat from the chaff. It also indicates how `cp` uses the mmap calls and how the shared libraries start up.

Table 3 Trace Output with Comments

Trace Output	Comments
% trace cp NewDocument Tuning	Use trace on a cp command
open ("/usr/lib/ld.so", 0, 04000000021) = 3	Get the shared library loader
read (3, ""..., 32) = 32	Read a.out header to see if dynamically linked
mmap (0, 40960, 0x5, 0x80000002, 3, 0) = 0xf77e0000	Map in code to memory
mmap (0xf77e8000, 8192, 0x7, 0x80000012, 3, 32768) = 0xf77e8000	Map in data to memory
open ("/dev/zero", 0, 07) = 4	Get a supply of zeroed pages
getrlimit (3, 0xf7fff8b0) = 0	Read the limit information
mmap (0xf7800000, 8192, 0x3, 0x80000012, 4, 0) = 0xf7800000	Map /dev/zero to the bss?
close (3) = 0	Close ld.so
getuid () = 1434	Get user id
getgid () = 10	Get group id
open ("/etc/ld.so.cache", 0, 05000000021) = 3	Open the shared library cache
fstat (3, 0xf7fff750) = 0	See if cache is up to date
mmap (0, 4096, 0x1, 0x80000001, 3, 0) = 0xf77c0000	Map it in to read it
close (3) = 0	Close it
open ("/usr/openwin/lib", 0, 01010525) = 3	LD_LIBRARY_PATH contains /usr/openwin/lib
fstat (3, 0xf7fff750) = 0	so look there first
mmap (0xf7802000, 8192, 0x3, 0x80000012, 4, 0) = 0xf7802000	
getdents (3, 0xf78000d8, 8192) = 1488	Get some directory entries looking for the right
getdents (3, 0xf78000d8, 8192) = 0	version of the library
close (3) = 0	Close /usr/openwin/lib
open ("/usr/lib/libc.so.1.6", 0, 032724) = 3	Get the shared libc
read (3, ""..., 32) = 32	Check its OK
mmap (0, 458764, 0x5, 0x80000002, 3, 0) = 0xf7730000	Map in the code

Table 3 Trace Output with Comments

Trace Output	Comments
mmap (0xf779c000, 16384, 0x7, 0x80000012, 3, 442368) = 0xf779c000	Map in the data
close (3) = 0	Close libc
close (4) = 0	Close /dev/zero
open (“NewDocument”, 0, 03) = 3	Finally! open input file
fstat (3, 0xf7fff970) = 0	Stat its size
stat (“Tuning”, 0xf7fff930) = -1 ENOENT (No such file or directory)	Try to stat output file
stat (“Tuning”, 0xf7fff930) = -1 ENOENT (No such file or directory)	But it’s not there
creat (“Tuning”, 0644) = 4	Create output file
mmap (0, 82, 0x1, 0x80000001, 3, 0) = 0xf7710000	Map input file
mctl (0xf7710000, 82, 4, 0x2) = 0	Madvise sequential access
write (4, “This is a test file for my paper”.., 82) = 82	Write out to new file
munmap (0xf7710000, 82) = 0	Unmap input file
close (3) = 0	Close input file
close (4) = 0	Close output file
close (0) = 0	Close stdin
close (1) = 0	Close stdout
close (2) = 0	Close stderr
exit (0) = ?	Exit program
%	

Timing

The C shell has a built-in `time` command that is used when benchmarking or tuning to see how a particular process is running.

```
% time man madvise
...
0.1u 0.5s 0:03 21% 0+168k 0+0io 0pf+0w
%
```

In this case 0.1 seconds of user cpu and 0.5 seconds of system cpu were used in 3 seconds elapsed time which accounted for 21% of the cpu. The growth in size of the process, the amount of i/o performed and the number of page faults and page writes are recorded. Apart from the times, the number of page faults is the most useful figure. In this case everything was already in memory from a previous use of the command.

The Effects Of Underlying Filesystem Type

Some programs are predominantly I/O intensive or may open and close many temporary files. SunOS has a wide range of filesystem types and the directory used by the program could be placed onto one of the following types.

UFS

The standard filesystem on disk drives is the Unix File System, which in SunOS 4.1 and on is the Berkeley Fat Fast Filesystem. If your files have more than a temporary existence then this will be fastest. Files that are read will stay in RAM until a RAM shortage reuses the pages for something else. Files that are written get sent out to disk but the file will stay in RAM until the pages are reused for something else. There is no special buffer cache allocation, unlike other Berkeley derived versions of Unix. SunOS and SVR4 both use the whole of memory to cache pages of code, data or I/O and the more RAM there is the better the effective I/O throughput will be. See the disk chapter for more info.

tmpfs

This is a real RAM disk filesystem type. Files that are written never get put out to disk as long as there is some RAM available to keep them in. If there is a RAM shortage then the pages end up being stored in the swap space. The most common way to use this is to un-comment the line in /etc/rc.local for `mount /tmp`. Some operations, such as file locking, are not supported in early versions of the tmpfs filesystem so applications that lock files in /tmp will misbehave. This is fixed in SunOS 4.1.2 and tmpfs is the default in Solaris 2.

```
# The following will mount /tmp if set up in /etc/fstab.
# If you want to use
# the anonymous memory based file system,
# have an fstab entry of the form:
#       swap    /tmp    tmp rw 0 0
# Make sure that option TMPFS is configured in the kernel
# (consult the System and Network Administration Manual).
#
mount /tmp
```

One side effect of this is that the free swap space can be seen using df.

```
% df /tmp
Filesystem            kbytes    used   avail capacity  Mounted on
swap                   15044     808   14236     5%    /tmp
```

NFS

This is a networked filesystem coming from a disk on a remote machine. It tends to have reasonable read performance but can be poor for writes and is slow for file locking. Some programs that do a lot of locking are unusable on NFS mounted filesystems. See the networking chapter for more information on tuning NFS performance.

References

- "tmpfs: A Virtual Memory File System, by Peter Snyder"

Databases & Configurable Systems

This chapter is concerned with tuning programs, such as databases, that the user is relying on in order to run his own application. The main characteristic is that these programs may provide a service to the system being tuned and they have sophisticated control or configuration languages.

Examples

Examples include relational databases such as Oracle, Ingres, Informix and Sybase which have large numbers of configuration parameters and an SQL based configuration language; CAD systems such as Autocad and Medusa; and Geographical Information Systems systems such as Smallworld GIS which have sophisticated configuration and extension languages.

Hire An Expert!

For serious tuning you either need to read all the manuals cover to cover and attend training courses or hire an expert for the day. The black box mentality of using the system exactly the way it came off the tape with all parameters set to default values will get you going but there is no point tuning the rest of the system if it spends 90% of its time inside a poorly configured database.

Use Sun's Database Excelerator Product

Sun has a version of SunOS tuned for use on systems with large amounts of memory running databases. It is called DBE - Database Excelerator and there are versions for each recent release of SunOS; DBE 1.2 for SunOS 4.1.2 and DBE 1.3 for SunOS 4.1.3. It is cheap at a few hundred pounds for media, manual and site licence, and it can dramatically improves the performance of databases, particularly with large numbers of concurrent users. If used on a system with less than 16 Mb of RAM it is likely to run more slowly than the standard SunOS since several algorithms have been changed to improve speed at the expense of more memory usage so 16 Mb is the minimum configuration.

Basic Tuning Ideas

Several times I have discovered untuned copies of Oracle so some basic recommendations on the first things to try may be useful. They apply to other database systems in principle.

Increasing Buffer Sizes

Oracle uses an area of shared memory to cache data from the database so that all oracle processes can access the cache. It defaults to about 400Kbytes but it can be increased to be bigger than the entire data set if needed. I would increase it to about 10% of the total RAM in the machine as a first try. There are ways of looking at the cache hit rate within Oracle so increase the size until the hit rate stops improving or the rest of the system starts showing signs of memory shortage. Avoiding unnecessary random disk I/O is one of the keys to database tuning.

Using raw disk rather than filesystems

You should reserve at least three empty disk partitions, spread across as many different disks and controllers as possible (but avoiding the *a* or *c* partition) when installing SunOS. You can then change the raw devices to be owned by oracle and when installing Oracle specify the raw devices rather than files in the usual filesystem as the standard data, log1 and log2 files. Filesystems incur more CPU overhead than raw devices and can be much slower for writes due to inode and indirect block updates. Two or three blocks in widely spaced parts of the disk must be written to maintain the filesystem, while only one block needs to be written on a raw partition. I have anecdotal reports of a doubling in database performance when moving from filesystems to raw partitions.

Balance The Load Over All The Disks

The log files should be on a separate disk from the data if possible. This is particularly important for databases that have a lot of update activity. It will also help to put indexes on their own disk or to split the database tables over as many disks as possible. The system disk is often lightly used and on a two disk system I would put the log files on the system disk and put the rest on its own disk. See “Load Monitoring And Balancing” on page 74.

Which Disk Partition To Use

If you use the first partition on a disk as a raw oracle partition then you will lose the disk's label. This can be recovered using the `format` command if you are lucky but you should make a filesystem, swap space or small unused partition at the start of the disk.

On Sun's 424Mb and 1.3Gb disks the first part of the disk is the fastest so a tiny first partition followed by a database partition covering the first half of the disk is recommended for best performance. See "ZBR Drives" on page 70 for more details and an explanation.

The Effect Of Indexes

When you look up an item in a database it must match your request against all the entries in a (potentially large) table. Without an index a full table scan must be performed and the database will read the entire table from disk in order to search every entry. If there is an index on the table the database will lookup the request in the index and it will know which entries in the table need to be read from disk. Some well chosen indexes can dramatically reduce the amount of disk I/O and CPU time required to perform a query. Poorly designed or untuned databases are often under-indexed.

Kernel

This chapter is concerned with variables that can be changed by a system administrator building or tuning a kernel. In SunOS 4.X the kernel must be re-compiled after tweaking a parameter file to increase table sizes but in Solaris 2 there is no need to re-compile the kernel, it is modified by changing /etc/system. The kernel algorithms have not changed much between SunOS 4.X and System V.4 or Solaris 2. The differences are noted as SVR4 changes if they are generic and as Solaris 2 where Solaris 2 is different to generic SVR4.

Later releases of SunOS 4.X have Solaris 1.X names which I have avoided for clarity. Part of Solaris 2.0 is known as SunOS 5.0 but this name is little used.

Buffer Sizes And Tuning Variables

The number of fixed size tables in the kernel has been reduced in each release of SunOS. Most are now dynamically sized or are linked to the maxusers calculation. The tuning required for each release varies as shown below.

Table 4 Tuning SunOS Releases

Release	Extra Tuning Required (apart from maxusers)
SunOS 4.0	Number of streams, number of mbufs
SunOS 4.0.1	Same as SunOS 4.0
SunOS 4.0.3	Add PMEGS patch tape, set handspread
SunOS 4.1	Add PMEGS patch tape or DBE-1.0, set handspread
SunOS 4.1.1	Add DBE-1.1, increase buffer cache
SunOS 4.1.2	Add DBE-1.2, add I/O patch 100575-02, increase maxslp
SunOS 4.1.3	Add DBE-1.3, Increase maxslp
Solaris 2.0	Increase paging parameters, check /etc/TIMEZONE

Maxusers In SunOS 4.X

The maxusers parameter is set in the kernel configuration file. Many parameters and kernel tables are derived from it. It is intended to be derived from the number of users on the system but this usually results in too small a value. It defaults to 16 for the sun4m architecture, but for other architectures it defaults to 8 for a GENERIC kernel and 4 for a GENERIC_SMALL. These values are suitable for single user workstations that are short of RAM but in most cases a substantial increase in maxusers is called for. A safe upper limit is documented in the SunOS 4.1.2 (about 100) and 4.1.3 (225) manuals and a very rough guideline would be to set to the number of Megabytes of RAM in the system for a workstation and twice that for an NFS server. Due to a shortage of kernel memory in SunOS 4.1.2 the safe upper limit is *reduced* to about 100 for systems that have large amounts of RAM since kernel memory is used to keep track of RAM as well as to allocate tables. The kernel base address was changed in 4.1.3 to allow a safe limit of 225 for any system.

Maxusers in Solaris 2

The effect of maxusers has not changed but it now defaults to 8 and is modified by placing commands in /etc/system e.g.

```
set maxusers=32
```

Be very careful with set commands in /etc/system, they basically cause automatic `adb` patches of the kernel so there is plenty of opportunity to completely screw up your system. The “Administering Security, Performance and Accounting in Solaris 2.0” manual states that NFS servers with 32Mb of RAM, up to four disks and up to ten logged in users maxusers should be set to 64. With more disks and users it should be set to 128.

Table 5 Default Settings for Kernel Parameters

Kernel Table	Variable	Default Setting
Callout	ncallout	16 + max_nprocs
Inode	ufs_ninode	max_nprocs + 16 + maxusers + 64
Name Cache	ncsize	max_nprocs + 16 + maxusers + 64

Table 5 Default Settings for Kernel Parameters

Kernel Table	Variable	Default Setting
Process	max_nprocs	10 + 16 * maxusers (must be under 4096)
Quota Table	ndquot	(maxusers * NMOUNT)/4 + max_nprocs
User Process	maxuprc	max_nprocs - 5

Directory Name Lookup Cache (dnlc)

This is sized using maxusers and a large cache size (ncsize above) significantly helps NFS servers which have lots of clients[1]. The command vmstat -s shows the DNLC hit rate. Directory names less than 14 characters long are cached and names that are too long are reported as well. A cache miss means that a disk I/O may be needed to read the directory when traversing the pathname components to get to a file. A hit rate of much less than 70% may need attention.

```
% vmstat -s
... lines omitted
    29600 total name lookups (cache hits 82% per-process)
          toolong 3077
```

Pstat in SunOS 4.X

The occupancy and size of some of these tables can be seen using the pstat -T command. This is for a SPARCstation 1 running SunOS 4.1.1 with maxusers set to 8.

```
% pstat -T
217/582 filesThe system wide open file table
166/320 inodesThe inode cache
 48/138 processesThe system wide process table
13948/31756 swapKilobytes of swap used out of the total
```

The pstat command only shows a few of the tables. Before SunOS 4.1 it showed another entry, confusingly also labelled files, which was in fact the number of streams. From SunOS 4.1 on the number of streams is increased dynamically on demand so this entry was removed.

1. See “Networks and File Servers: A Performance Tuning Guide

Inode Cache

The inode cache is used whenever an operation is performed on an entity in the UFS filesystem. The inode read from disk is cached in case it is needed again. An IO benchmark which opened and closed a large number of files could be tuned by increasing the size of the inode cache such that it produced results showing a slow SCSI disk performing at an apparent rate of over 10 Mbytes per second. Since the benchmark was claimed to be representative of common disk usage patterns a large inode cache can help I/O intensive programs significantly. Other versions of Unix (apart from SVR4), which use a different virtual memory and I/O setup, produced results for this benchmark that were an order of magnitude slower. The inode cache seems to grow dynamically at a fairly slow rate if it is used intensively and increasing maxusers provides a much larger initial size for the cache.

Buffer Cache

The buffer cache is used to cache all UFS disk I/O in SunOS 3 and BSD Unix. In SunOS 4 and SVR4 it is used to cache inode and cylinder group related disk I/O only. It is set via the kernel variable `nbuf` and is usually set to around 32. It should be increased to 64 for machines with up to four disks and 112 for machines with more than four disks[1]. For SunOS 4.1.2 and 4.1.3 it is no longer necessary to set `nbuf` as it is sized automatically.

Solaris 2.0 Tuning

There is a manual section called "Administering Security, Performance and Accounting in Solaris 2.0". Read it!

There is a bug in the C library implementation of `localtime` which means that some timezone settings cause the `zoneinfo` file to be reread on every call to `localtime` or `mktime`. This occurs when the symbolic `timezone` names are used, e.g. GB-Eire or US/Pacific. The work-around is to use a descriptive timezone name such as GMT0BST (rather than GB-Eire) or PST8PDT (rather than US/Pacific) where the zone can be determined directly rather than using the `zoneinfo` data. The dates that summer time start/stop may be inaccurate

1. Details are in "Tuning the SPARCserver 490 for Optimal NFS Performance, February 1991

but some applications like Calendar Manager run many times faster and a simple `ls -l` is noticeably quicker. This is fixed in Solaris 2.1 by reading the zoneinfo file once and caching it for future use.

Solaris 2.1 Performance

Solaris 2.1 has had a large amount of performance tuning work done on it. It is substantially faster than Solaris 2.0 and performs at a similar level to SunOS 4.1.3 on uniprocessor machines. On multiprocessor machines Solaris 2.1 is often faster than SunOS 4.1.3. See the chapter on multiprocessors.

Paging, MMU Algorithms And PMEGS

Sar in Solaris 2

This utility has a huge number of options and some very powerful capabilities. One of its best features is that you can log its full output in date-stamped binary form to a file. You can even look at a few selected measures then go back to look at all the other measures if you need to. It is described in full in "Administering Security, Performance and Accounting in Solaris 2.0" and in "System Performance Tuning, Mike Loukides, O'Reilly".

Vmstat

The paging activity on a Sun can be monitored using vmstat. SunOS 4.1.1 output is shown below. Solaris 2.0 reuses the avm field to show free swap space.

```
% vmstat 5
 procs     memory              page              disk       faults     cpu
 r b w   avm   fre  re at  pi  po  fr  de  sr s0 s1 d2 s3  in  sy  cs us sy id
 0 0 0     0  1072   0  2   4   1   3   0   1  1  0  0  0  43 217  15  7  4 89
 1 0 0     0   996   0  4   0   0   0   0   0  4  0  0  0 112 573  25 11  6 83
 0 0 0     0   920   0  0   0   0   0   0   0  0  0  0  0 178 586  43 25  9 67
 0 0 0     0   900   0  0   0   0   0   0   0  3  0  0  0 127 741  30 13  6 81
 0 0 0     0   832   0  0   4   0   0   0   0  0  0  0  0 171 362  44  7  6 87
 0 0 0     0   780   0  0  72   0   0   0   0 13  0  5  0 158 166  45  3  8 89
 0 3 0     0   452   0  0 100   0  76   0  47 20  0  3  0 200 128  79  6 11 83
 0 0 0     0   308   0  2  28   0  20   0  15  2  0  1  0  69  50  28  3  4 93
 0 0 0     0   260   0  3   8   0  24   0  12  0  0  1  0  44 102  25  5  4 91
 0 0 0     0   260   0  0   0   0   0  16   3  3  0  1  0  42  68  12  3  5 92
```

Unfortunately the vmstat manual page, even in SunOS 4.1.1 is misleading in some respects so clarification of some fields is in order. See the manual page.

Table 6 Vmstat fields explained

Field	Explanation
Memory	Report on usage of real memory.
avm or swap	**Active virtual memory is a historical measure that is always set to zero. swap shows the free virtual memory in Kbytes for Solaris 2.0**
fre	Free real memory in Kbytes
Page	Report information about page faults and paging activity. The information on each of the following activities is averaged each five seconds, and given in units per second.
re	pages reclaimed from the free list
at	number of attaches to pages already in use by other processes
pi	kilobytes per second paged in
po	kilobytes per second paged out
fr	kilobytes freed per second
de	artificial memory deficit set during swap out to prevent immediate swapin
sr	pages scanned by clock algorithm, per-second

Looking at the above log of vmstat it can be seen that some CPU activity in the first few entries was entirely memory resident. This was followed by some paging and as the free memory dropped below 256K the scanning algorithm woke up and started to look for pages that could be reused to keep the free list above its 256K minimum. The five second average shows that the result of this is a free list at around 300K. This is typical for a Sun that has been running for a while and is not idle. If the free list is consistently greater than 1 Mb on pre 4.1.1 SunOS releases you may have a PMEG problem!

The Sun-4 MMU - sun4, sun4c, sun4e kernel architectures

Older Sun machines use a “Sun-4” hardware memory management unit which acts as a large cache for memory translations. It is much larger than the MMU translation cache found on more recent systems but the entries in the cache, known as PMEGs are larger and take longer to load. A PMEG is a Page Map Entry Group, a contiguous chunk of virtual memory made up of 32 or 64

physical 8Kb or 4Kb page translations. A SPARCstation 1 has a cache containing 128 PMEGS so a total of 32Mb of virtual memory can be cached. The number of PMEGS on each type of Sun is shown in Table 5-2.

Table 7 Sun-4 MMU Characteristics

Processor type	Page Size	Pages/PMEG	PMEGS	Total VM	Contexts
SS1(+), SLC, IPC	4 Kb	64	128	32 Mb	8
ELC	4 Kb	64	128	32 Mb	8
SPARCengine 1E	8 Kb	32	256	64 Mb	8
IPX	4 Kb	64	256	64 Mb	8
SS2	4 Kb	64	256	64 Mb	16
Sun 4/110, 150	8 Kb	32	256	64 Mb	16
Sun 4/260, 280	8 Kb	32	512	128 Mb	16
SPARCsystem 300	8 Kb	32	256	64 Mb	16
SPARCsystem 400	8 Kb	32	1024	256 Mb	64

In SunOS 4.0.3 and SunOS 4.1 there were two independent problems which only came to light when Sun started to ship 4Mbit DRAM parts and the maximum memory on a SPARCstation 1 went up from 16 Mb to 40Mb. Memory prices also dropped and many more people loaded up their SPARCstation 1,1+, SLC or IPC with extra RAM. When the problem was discovered a patch was made for SunOS 4.0.3 and SunOS 4.1. The patch is incorporated into DBE 1.0 and later and SunOS 4.1.1 and later as standard.

If you run within the limits of the MMU's 128 PMEG entries the processor runs flat out; faster than other MMU architectures in fact. When you run outside the limits of the MMU the problems occur.

PMEG reload time problem

When a new PMEG was needed the kernel had to get information from a number of data structures and process it to produce the values needed for the PMEG entry. This took too long and one of the symptoms of PMEG thrashing was that the system CPU time is very high, often over 50% for no apparent reason. The cure is to provide a secondary, software cache for the completed PMEG entries which can be resized if required. If a PMEG entry is already in

the software cache then it is block copied into place. The effect is that the reload time is greatly reduced and the amount of system CPU used drops back to more reasonable levels.

PMEG Stealing

In order to load a new PMEG the kernel had to decide which existing entry to overwrite. The original algorithm used made the problem much worse since it often stole a PMEG from an active process, which would then steal it back again, causing a thrashing effect. When a PMEG was stolen its pages were put onto the free list. If every PMEG was stolen from a process then every page would be on the free list and the system would decide to swap out the process. This gave rise to another symptom of PMEG thrashing, a large amount of free memory reported by vmstat and a lot of swapping reported by `vmstat -S` even though there was no disk I/O going on. The cure is to use a much better algorithm for deciding which PMEG to reuse and to stop putting pages on the free list when a PMEG is stolen.

Problem Solved

There are some performance graphs in the "SPARCstation 2 Performance Brief" which show the difference between SunOS 4.1 and SunOS 4.1.1 using a worst case test program. For a SPARCstation IPC the knee in the curve which indicates the onset of PMEG thrashing is at 16 Mb for SunOS 4.1 and around 60 Mb for SunOS 4.1.1. Beyond that point SunOS 4.1 hits a brick wall at 24 Mb while the IPC is degrading gracefully beyond 80 Mb with SunOS 4.1.1.

The SPARCstation 2, which has twice as many PMEGs, is flat beyond 80 Mb with no degradation.

If you are called upon to tune a system that seems to have the symptoms described above, is running SunOS 4.0.3 or SunOS 4.1 and you cannot upgrade to SunOS 4.1.1, then you should contact the Sun Answer Centre to get hold of the PMEGS patch.

Context s

Table 7 on page 37 shows the number of hardware contexts built into each machine. A hardware context can be though of as a tag on each PMEG entry in the MMU which indicates which process that translation is valid for. This allows the MMU to keep track of the mappings for 8, 16 or 64 processes in the

MMU, depending upon the machine. When a context switch occurs, if the new process is assigned to one of the hardware contexts, then some of its mappings may still be in the MMU and a very fast context switch can take place. For up to the number of hardware contexts available this scheme is more efficient than a more conventional TLB based MMU. When the number of processes trying to run exceeds the number of hardware contexts the kernel has to choose one of the hardware contexts to be reused and has to invalidate all the PMEGS for that context and load some PMEGS for the new context. The context switch time starts to degrade gradually and probably becomes worse than a TLB based system when there are more than twice as many active processes as contexts. This is one reason why a SPARCserver 490 can handle many more users in a timesharing environment than the (apparently faster) SPARCstation 2, which would spend more time in the kernel shuffling the PMEGS in and out of its MMU. There is also a difference in the hardware interface used to control the MMU and cache, with more work needing to be done in software to flush a context on a SPARCstation 1 and higher level hardware support on a SPARCstation 2 or SPARCserver 400.

No new machines are being designed to use this type of MMU, but it represents the bulk of the installed base.

The SPARC Reference MMU - sun4m kernel architecture

Recently Sun's have started to use the SPARC Reference MMU which has an architecture that is similar to many other MMU's in the rest of the industry.

Table 8 SPARC Reference MMU Characteristics

Processor Types	Page Sizes	TLB Entries	Contexts	Total VM
Ross 6002 Modules Cypress 604 and 605 MMU's SPARCserver 600 -120 and -140 Tadpole SPARCbook 1	4 Kb 256Kb 16Mb	64	4096	256Kb 16Mb 1024Mb
Texas Instruments SuperSPARC SPARCserver 600 -41, -52 and -54 SPARCstation 10 -30, -41, -52,and -54	4 Kb 256Kb 16Mb	64	65536	256Kb 16Mb 1024Mb
Fujitsu SPARClite embedded CPU	4 Kb 256Kb 16Mb	32	256	128Kb 8Mb 512Mb
Texas Instruments MicroSPARC SPARCclassic and SPARCstation LX	4Kb 256Kb 16Mb	32	64	128Kb 8Mb 512Mb

There are four current implementations, the Cypress uniprocessor 604 and multiprocessor 605 MMU chips, the MMU that is integrated into the SuperSPARC chip, the Fujitsu SPARClite and the highly integrated MicroSPARC (which I do not have full details of yet).

In this case there is a small fully associative cache for address translations (a Translation Lookaside Buffer or TLB) which typically has 64 entries that map one contiguous area of virtual memory each. These areas are usually a single 4Kb page but future releases of Solaris 2 may be optimised to use the 256Kb or 16Mb modes in certain cases. Each of the 64 entries has a tag that indicates what context it belongs to. This means that the MMU does not have to be flushed on a context switch. The tag is 12 bits on the Cypress/Ross MMU and 16 bits on the SuperSPARC MMU, giving rise to a much larger number of hardware contexts than the Sun-4 MMU so that MMU performance is not a problem when very large numbers of users or processes are present.

The primary difference from the Sun-4 MMU is that TLB entries are loaded automatically by table walking hardware in the MMU. The CPU stalls for a few cycles waiting for the MMU but unlike many other TLB based MMU's or the Sun-4 MMU the CPU does not take a trap to reload the entries itself. The kernel builds a table in memory that contains all the valid virtual memory mappings and loads the address of the table into the MMU once at boot time. The size of this table is large enough for most purposes but a system with a large amount of active virtual memory can cause a shortage of page table entries with similar effects to a PMEGS shortage. The solution is simply to increase the amount of kernel memory allocated to this purpose. The kernel variable `npts` controls how many are allocated. It is calculated depending upon the amount of physical memory in the system but can be set explicitly by patching `npts`. On a 64Mb SPARCserver 600 running a GENERIC 4.1.2 kernel `npts` is set to 1065 which seems on the low side to me. Both DBE 1.2 and SunOS 4.1.3 set `npts` much higher.

The Sun4-MMU based systems can cache sufficient virtual memory translations to run programs many Mb in size with no MMU reloads. When the MMU limits are exceeded there is a large overhead. The SPARC Reference MMU only caches 64 pages of 4Kb at a time in normal use for a total of 256Kb. The MMU is reloading continuously but it has an exceptionally fast reload.

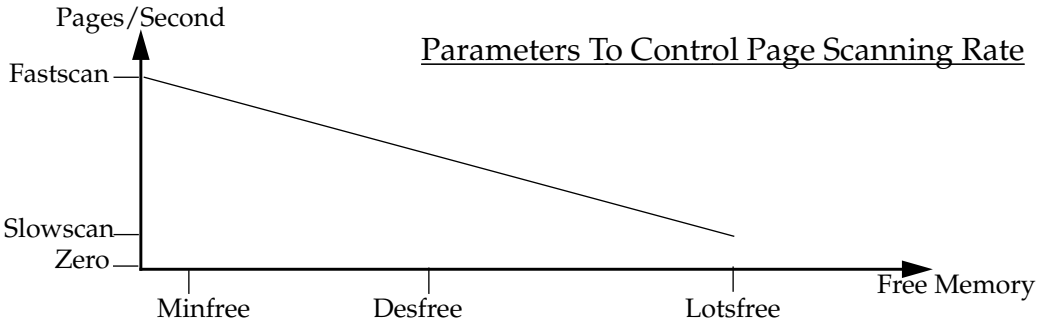
The Paging Algorithm

When new pages are allocated from the free list there comes a point when the system decides that there is no longer enough free memory (less than `lotsfree`) and it goes to look for some pages that haven't been used recently to add to the free list. At this point the pagedaemon is woken up. The system also checks the size of the free list four times per second and may wake up the pagedaemon. After a wakeup the pagedaemon is scheduled to run as a process inside the kernel and assumes that it runs four times per second so it calculates a scan rate then divides by four to get the number of pages to scan before it goes back to sleep.

The pagedaemon's scanning algorithm works by regarding all the pagable RAM in order of its physical address as if it was arranged in a circle. Two pointers are used like the hands of a clock and the distance between the two hands is controlled by `handspread`. When there is a shortage of free pages (less than `lotsfree`) the hands start to move round the clock at a slow rate `(slowscan)` which increases linearly to a faster rate `(fastscan)` as free memory tends to zero. If the pages for each hand are not locked into memory,

on the free list or otherwise busy then a flag which is set every time the pages are referenced is examined and if they have not been referenced the pages can be freed. The first hand then clears the referenced flag for its page so that when the second hand gets round to the page it will be freed unless it has been referenced since the first hand got there. If the freed page contained modified data it is paged out to disk. If there is a lot of memory in the system then the chances of an individual page being referenced by a CPU in a given time span are reduced so the spread of the hands must be increased to compensate.

If the shortage of memory gets worse (less than `desfree`), there are two or more processes in the run queue, and it stays at that level for more than 30 seconds then swapping will begin. If it gets to a minimum level (`minfree`) swapping starts immediately. If after going twice through the whole of memory there is still a shortage, the swapper is invoked to swap out entire processes. The algorithm limits the number of pages scheduled to be paged out to 40 per second (`maxpgio`) since this is a good figure for the number of random I/Os per second on a single disk. If you have swap spread across several disks then increasing `maxpgio` may improve paging performance and delay the onset of swapping. Note that `vmstat po` reports the number of kilobytes per second paged out which can be compared to `maxpgio * pagesize`.



Kernel Variables To Control Paging

minfree

This is the absolute minimum memory level that can be tolerated by the system. If (freemem - deficit) is less than `minfree` the system will immediately swap processes out rather than paging. It is often set to 8 pages and clamped at desfree/2. The SunOS 4 sun4m kernel has a higher default and Solaris 2.1 scales the value.

desfree

This represents a *desperation* level, if free memory stays below this level for more than 30 seconds then paging is abandoned and swapping begins. It is often set to 25 pages and is clamped to (total memory)/16. The SunOS 4 sun4m kernel has a higher default and Solaris 2.1 scales the value.

lotsfree

This is the memory limit that triggers the page daemon to start working if free memory drops below it. It is often set to 64 pages and is clamped at (total memory)/8. The sun4m kernel has a higher default of around 500 pages and Solaris 2.1 scales the value.

fastscan

This is the number of pages scanned per second by the algorithm when there is `minfree` available and it is often set to 1000. There is a linear ramp up from `slowscan` to `fastscan` as free memory goes from lotsfree to zero. The SunOS 4.X sun4m kernel sets it to (total memory in pages)/2. The Solaris 2.1 kernel also scales up by the amount of RAM in the system.

slowscan

This is the number of pages scanned per second by the algorithm when there is just under `lotsfree` available and it is often set to 100. There is a linear ramp up from `slowscan` to `fastscan` as free memory goes from `lotsfree` to zero. The sun4m kernel has a higher default of `fastscan`/10.

maxpgio

This is the maximum number of page out I/O operations per second that the system will schedule. The default is 40 pages per second, which is set to avoid saturating random access to a single 3600 rpm (60 rps) disk at two-thirds of the rotation rate. It can be increased if more or faster disks are being used for the swap space. Many systems now have 5400 rpm (90 rps) disks, see Table 14 on page 69 for disk specifications.

handspread

Handspread is set to (total memory)/4, but is increased to be at least as big as fastscan which makes it (total memory)/2 on sun4m machines.

Unfortunately, in the old days of Berkeley Unix, when an 8 Mb VAX 750 was a big machine, the code that set up handspread clamped it to a maximum of 2 Mb. This code was fixed in SunOS 4.1.1 so handspread needs to be patched on any machine that has much more than 8 Mb of RAM and is running SunOS 4.1 or before.

The default value in `/vmunix` is 0 which makes the code calculate its own value. In this case, just under 4 Mb since this machine has 16Mb of RAM and is running SunOS 4.1.1.

```
# adb -k -w /vmunix /dev/mem
physmem ff3
handspread?X
_handspread:
_handspread:    0
handspread/X
_handspread:
_handspread:    3e8000
```

If the default value is patched using adb then that value is used rather than worked out from the available RAM. This is how it can be fixed in releases of SunOS prior to 4.1.1, using `handspread?W0x3e8000` to patch the value into `/vmunix` and `handspread/W0x3e8000` to patch the value in memory.

The effect of the problem is that pages that are actively being used are more likely to be put on the free list if handspread is too small, causing extra work for the kernel. If the system has a lot of memory installed an stop/go effect can occur where the scanning algorithm will be triggered by a shortage of memory and it will free every page it finds until there is masses of free memory then it

will go to sleep again. On a properly configured system it should be trickling around continuously at a slow rate tidying up the pages that haven't been referenced for a while.

Swapping Out And How To Get Back In

The kernel keeps track of the ten biggest processes according to their resident set (RSS). For swapping it selects the four largest and swaps out the oldest or a process that has been sleeping longer than `maxslp`. The deficit (`vmstat de`) is increased to prevent the process from swapping back in immediately.

The first processes to be swapped back in are the smallest ones that have been asleep for longest and are not running at low priority. A process will be swapped back in when half of the number of pages it freed on swapout are available, although only the basic few pages are swapped in in one go and the rest are paged in as needed.

Sensible Tweaking

The default settings for most machines were derived to handle machines with at most 8 Mbytes of RAM and around one MIP performance. These settings were tinkered with for the SPARCserver 600MP (the initial sun4m machine) and have been overhauled for Solaris 2.1. The suggestions below are a mixture of theory, informed guesswork and trial and error testing. If you try any of the suggestions and you *measure* a significant improvement (or reduction) in performance please let me know.

Increasing Lotsfree

An interactive timesharing system will often be continuously starting new processes, interactive window system users will keep opening new windows. This type of system can benefit from an increase in the paging variables. A pure database server, compute server or single application CAD workstation may start-up then stay very stable, with a long time between process start-ups. This type of system can use the default values.

Taking a window system example: When a new process starts it consumes free memory very rapidly before reaching a steady state so that the user can begin to use the new window. If the process requires more than the current free memory pool then it will get part way through its start-up then the paging

algorithm will take over the CPU and look for more memory at up to `fastscan` rates and may trigger a swap out. When the memory has been found the process continues. This interruption in process start-up time is very apparent to the user as poor interactive response. To improve the interactive response `lotsfree` can be increased so that there is a large enough pool of free memory for most processes to start-up without running out. If `slowscan` is decreased then after they have started up the page daemon will gently find some more free memory at `slowscan` rates until it reaches `lotsfree` again.

In the past the SunOS 4.X default free pool of 64 8Kb pages provided 512Kb, when the page size went to 4Kb for the desktop sun4c kernel architecture the free pool went to 256Kb. On the sun4m kernel architecture it is set to (total available memory)/32 which defaults to just under 2048Kb on a 64Mb SPARCserver 600. The problems occur on the sun4c machines since the move from SunView to OpenWindows/X11 seemed to coincide with an increase in start-up memory requirements to more than 256Kb for most applications. I recommend increasing the `lotsfree` parameter to (total memory)/32 on all sun4c machines. `Desfree` should be left at a low level to inhibit swapping. You can try this out by timing a process start-up in one window[1], monitoring vmstat in another window and tweaking `lotsfree` with adb in a third window.

If you are running a large stable workload and are short of memory, it is a bad idea to increase `lotsfree` because more pages will be stolen from your applications and put on the free list. You want to have a small free list so that all the pages can be used by your applications.

Problems can occur if very large processes are swapped out on systems with very large amounts of memory configured since the amount of free memory maintained by lotsfree may not be enough to swap the process back in again promptly. Swapping can be totally disabled if this helps.

Changing Fastscan and Slowscan

The `fastscan` parameter is set to 1000 pages per second in SunOS 4.X machines apart from the sun4m architecture where it is set to (total memory pages)/2 pages. (e.g. about 4000 on a 32Mb machine and about 125000 on a 1Gb machine). `Slowscan` is set to 100 or on sun4m, `fastscan`/10 (e.g. about

1. For OpenWindows applications, “time toolwait application” will show the elapsed start-up time.

400 on a 32Mb machine and about 12500 on a 1Gb machine). These very high sun4m scan rates consume excessive system CPU time and `slowscan` in particular should be set quite low so that the number of pages scanned at the slowest rate doesn't immediately cause a paging I/O overload. `Fastscan` should probably be halved to (total memory pages)/4. The ramp up from slowscan to fastscan as free memory drops will be steep so there is no need to increase slowscan from the traditional level of 100 as memory is added.

Solaris 2.0 overcompensates for this and sets `fastscan` to 200 and `slowscan` to 100. These parameters are set higher and scaled automatically depending on the amount of installed memory in Solaris 2.1.

Disabling maxslp

If a process has been sleeping for more than 20 seconds then it is very likely to be swapped out. This means that the update process, which sync's the disks every 30 seconds is continually being swapped in and out since it tends to sleep for 30 seconds. It also means that clock icons swap in every minute to update the time. This concept seems to generate more overhead than it saves and it can be disabled by setting `maxslp` to 128. The kernel only keeps track of process sleep time in a single byte and it stops counting at 127 seconds. The concept of `maxslp` has gone away in Solaris 2 so there is no need to increase it and some encouragement that this is a good thing to do in SunOS 4.X. Large timesharing MP servers running SunOS 4.1.2 or 4.1.3 can be helped significantly due to a reduction in unnecessary system time overhead. The current values for all processes can be seen using pstat -p (in the SLP column) and information on a single process can be seen using pstat -u PID (where PID is the process ID). The pstat man page is helpful but intimate knowledge of kernel internals is assumed in too many cases.

SunOS 4.X "Standard" Tweaks For 32Mb Sun4c

The following will change the current operating copy *and* permanently patch /vmunix for the next reboot. If you don't notice any improvement you should stay with the default values. The right hand column is commentary only and I have omitted some of adb's output. When adb starts it prints out the value of `physmem` in hex, which is the total number of pages of physical memory. A few pages that are probably used by the boot prom are not included and the kernel memory should really be subtracted from `physmem` to get the total amount of pagable memory. Control-D exits adb.

The settings are based on disabling `maxslp`, setting `lotsfree` to (total memory)/32, increasing `maxpgio` to 2/3 of a single 4500 rpm, 75rps (424Mb) swap disk, and increasing `fastscan` to `physmem`/4. These values should be scaled according to the configuration of the machine being tuned.

```
# cp /vmunix /vmunix.notpatched
# adb -k -w /vmunix /dev/mem
physmem 1ffd       physmem is the total number of physical memory pages in hex
maxslp?W0x80       disable maxslp in /vmunix
maxslp/W0x80       disable current operating maxslp
lotsfree?W0x100    set lotsfree to 256 pages - 1 Mbyte in /vmunix
lotsfree/W0x100    set current lotsfree to 1 Mbyte
maxpgio?W0x32      set maxpgio to 50 pages/sec in /vmunix
maxpgio/W0x32      set current maxpgio to 50 pages/sec
fastscan?W0x800    set fastscan to 2048 pages/sec in /vmunix
fastscan/W0x800    set current fastscan to 2048 pages/sec
```

SunOS 4.1.X "Standard" Tweaks for 128Mb Sun4m

In this case the scan rate variables need to be modified to reduce `slowscan` to about 100 and to halve `fastscan`. Twin 5400 rpm, 90 rps (1.3Gb) swap disks are assumed so `maxpgio` can be set to 2*(90*2/3) = 120.

```
# cp /vmunix /vmunix.notpatched
# adb -k -w /vmunix /dev/mem
physmem 3ffd       physmem is the total number of physical memory pages in hex
maxslp?W0x80       disable maxslp in /vmunix
maxslp/W0x80       disable current operating maxslp
lotsfree?W0x400    set lotsfree to 1024 pages - 4 Mbyte in /vmunix
lotsfree/W0x400    set current lotsfree to 4 Mbyte
maxpgio?W0x78      set maxpgio to 120 pages/sec in /vmunix
maxpgio/W0x78      set current maxpgio to 120 pages/sec
fastscan?W0x2000   set fastscan to 8192 pages/sec in /vmunix
fastscan/W0x2000   set current fastscan to 8192 pages/sec
slowscan?W0x64     set slowscan to 100 pages/sec in /vmunix
slowscan/W0x64     set current slowscan to 100 pages/sec
```

Solaris 2.0 "Standard" Tweaks for 32Mb Sun4c

The following will change the current operating copy only, this is useful for experimenting but be careful that you keep `lotsfree` > `desfree` at all times!

```
# adb -k -w /dev/ksyms /dev/mem
physmem 1ffd       physmem is the total number of physical memory pages in hex
fastscan/W0x800    set fast scan rate to 1024 pages/second
lotsfree/W0x100    set current lotsfree to 1 Mbyte
maxpgio/W0x32      set current maxpgio to 50 pages/sec
```

The following commands should be placed at the end of /etc/system and they will be automatically applied at the next reboot.

```
set maxusers = 32      # increase kernel table sizes
set fastscan = 0x800   # set fast scan rate to 1024 pages/second
set lotsfree = 0x100   # set current lotsfree to 256 pages - 1 Mbyte
set maxpgio  = 0x32    # set current maxpgio to 50 pages/sec
```

Solaris 2.1

The way that paging parameters are chosen has been reworked in Solaris 2.1, it seems that “sensible” values are derived for most system configurations so tweaking is probably not required.

How many nfsds?

The NFS daemon `nfsd` is used to service requests from the network and a number of them are started so that a number of outstanding requests can be processed in parallel. Each `nfsd` takes one request off the network and passes it onto the I/O subsystem, to cope with bursts of NFS traffic a large number of `nfsds` should be configured, even on low end machines. All the `nfsds` run in the kernel and do not context switch in the same way as user level processes so the number of hardware contexts is not a limiting factor (despite folklore to the contrary!). On a dedicated NFS server between 40 and 60 `nfsds` should be configured. If you want to “throttle back” the NFS load on a server so that it can do other things this number could be reduced. If you configure too many `nfsds` some may not be used, but it is unlikely that there will be any adverse side effects.

Configuring Out Unused Devices

SunOS 4.X

This is one of the most obvious tuning operations for systems that are running very short of memory. It is particularly important to configure 8Mb diskless SPARCstation ELC's to have as much free RAM as possible since it will be used to cache the NFS accesses. A diskless machine can have many filesystem types and devices removed (including the nonexistent floppy on the ELC) which can free up over 500Kbytes. The DL60 configuration file with all framebuffer types except mono removed and with TMPFS added makes a good DL25 kernel for the ELC.

Solaris 2

There is no need to configure the kernel in Solaris 2 since it is dynamically linked at run time. All the pieces of the kernel are stored in a /kernel directory and the first time a device or filesystem type is used it is loaded into memory. The kernel can unload some of the devices if they are not being used.

The boot sequence builds a tree in of the hardware devices present in the system, which can be viewed with the `prtconf` command. This is mirrored in the /devices and /dev directories and after hardware changes are made to the system these directories must be reconfigured using `boot -r`. See also the `tapes` and `disks` commands that allow you to configure a system manually.

References

- "Administering Security, Performance and Accounting in Solaris 2.0"
- "Building and Debugging SunOS Kernels, by Hal Stern"
- "SPARCstation 2 Performance Brief"
- "SunOS System Internals Course Notes"
- "System Performance Tuning, Mike Loukides, O'Reilly"

Memory

Performance can vary even between machines that have the same CPU and clock rate if they have different memory systems. The interaction between caches and algorithms can cause problems. Unless you can choose a different machine to run on an algorithm change may be called for to work around the problem. This chapter provides information on the various SPARC platforms so that application developers can understand the implications of differing memory systems. This section may also be useful to compiler writers.

Cache Tutorial

Why Have a Cache?

Historically the memory systems on Sun machines provided equal access times for all memory locations. This was true on the Sun2 and was true for data accesses on the Sun3/50 and Sun3/60.

It was achieved by running the entire memory system as fast as the processor could access it. On Motorola 68010 and 68020 processors four clock cycles were required for each memory access. On a 10MHz Sun2 the memory ran at a 400ns cycle time and on a 20MHz Sun3/60 it ran at a 200ns cycle time.

Unfortunately, although main memory (known as DRAM) has increased in size, its speed has only increased a little. Today, processors rated around ten times the speed of a Sun3/60 run at a higher clock rate and access memory in a single cycle. The 40MHz SPARCstation 2 requires 25ns cycle times to keep running at full speed. The DRAM used is rated at 80ns *access time* which is less than a full single cycle of maybe 140ns. The DRAM manufacturers have added special modes that allow fast access to a block of consecutive locations after an initial address has been loaded. These *page mode* DRAM chips provide a block of data at the 50ns rate required for an SBus block transfer into cache on the SPARCstation 2, after a short delay to load up an address.

The CPU cache is an area of fast memory made from static RAM or SRAM. This is too expensive and would require too many chips to be used for the entire memory system so it is used in a small block of perhaps 64Kb. It can operate at the full speed of the CPU (25ns @ 40MHz) in single cycle reads and writes and it transfers data from main DRAM memory using page mode in a block of typically 16 or 32 bytes at a time. Hardware is provided to keep track of the data in the cache and to copy it into the cache when required.

More advanced caches can have multiple levels and can be split so that instructions and data use separate caches. The SuperSPARC with SuperCache chipset used in the SPARCstation 10 model 41 has a 20Kbyte instruction cache with 16 Kbyte data cache and these are loaded from a second level 1Mbyte combined cache that loads from the memory system. Simple caches will be examined first before we look at the implications of more advanced configurations.

Cache line and size effects

A SPARCstation 2 has its 64Kb of cache organized as 2048 blocks of 32 bytes each. When the CPU accesses an address the cache controller checks to see if the right data is in the cache, if it is then the CPU loads it without stalling. If the data needs to be fetched from main memory then the CPU clock is effectively stopped for 24 or 25 cycles while the cache block is loaded. The implications for performance tuning are obvious. If your application is accessing memory very widely and its accesses are *missing t*he cache rather than *hitting* the cache then the CPU may spend a lot of its time stopped. By changing the application you may be able to improve the *hit rate a*nd get a worthwhile performance gain. The 25 cycle delay is known as the *miss cost* and the effective performance of an application is reduced by a large miss cost and a low hit rate. The effect of context switches is to further reduce the cache hit rate as after a context switch the contents of the cache will need to be replaced with instructions and data for the new process.

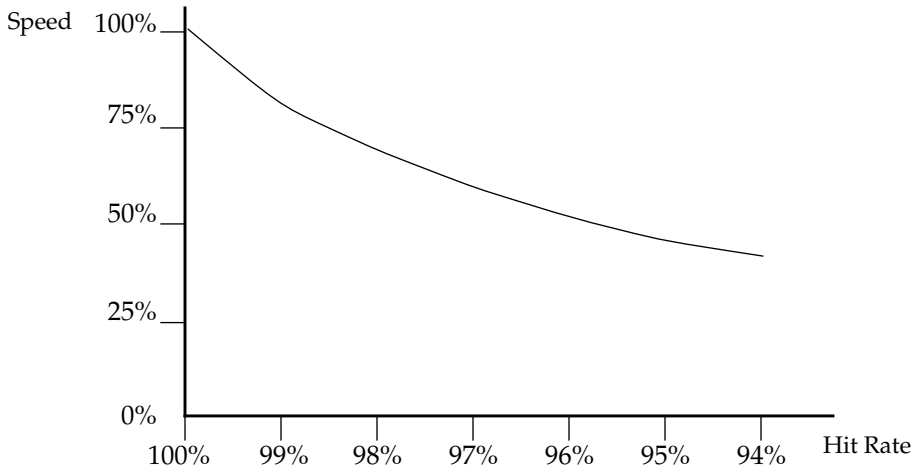
Applications are accessing memory on almost every cycle since most instructions take a single cycle to execute and the instructions must be read from memory, data accesses typically occur on 20-30% of the cycles. The effect

of changes in hit rate for a 25 cycle miss cost are shown below. in both tabular and graphical forms. A 25 cycle miss cost implies that a hit takes one cycle and a miss takes 26 cycles.

Table 9 Application speed changes as hit rate varies with a 25 cycle miss cost

Hit Rate	Hit Time	Miss Time	Total Time	Performance
100%	100%	0%	100%	100%
99%	99%	26%	125%	80%
98%	98%	52%	150%	66%
96%	96%	104%	200%	50%

Figure 1 Application speed changes as hit rate varies with a 25 cycle miss cost



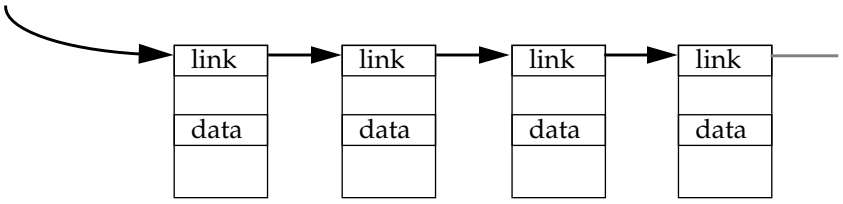
There is a dramatic increase in execution time as the hit rate drops. Although a 96% hit rate sounds quite high you can see that the program will be running at half speed. Many small benchmarks like Dhrystone run at 100% hit rate. The SPEC92 benchmark suite runs at a 99% hit rate given a 1Mbyte cache[1]. It isn't that difficult to do things that are bad for the cache however so it is a common cause of performance problems.

1. "The SuperSPARC Microprocessor Technical White Paper"

A Problem With Linked Lists

In "Algorithms" on page 13 I mentioned a CAD application that traversed a large linked list. Lets look at this in more detail and assume that the list has 5000 entries[1]. Each block on the list contains some data and a link to the next block. If we assume that the link is located at the start of the block and that the data is in the middle of a 100 byte block than the effect on the memory system of chaining down the list can be deduced.

Figure 2 Linked List Example



The code to perform the search is a tight loop shown in Figure 3. This code fits in seven words, one or two cache lines at worst, so the cache is working well for code accesses. Data accesses occur when the link and data locations are read. If the code is simply looking for a particular data value then these data accesses will be happening every few cycles. They will never be in the same cache line so every data access will cause a 25 cycle miss which will read in 32 bytes of data when only 4 bytes were wanted. Also there are only 2048 cache lines available so after 1024 blocks have been read in the cache lines must be reused. This means that a second attempt to search the list will find that the start of the list is no longer in the cache.

The only solution to this problem is an algorithmic change. The problem will occur on any of the current generation of high performance computer systems. In fact the problem gets worse as the processor gets faster since the miss cost will tend to increase due to the difference in speed between the CPU and cache clock rate and the main memory speed.

1. See also "The Classic Cache Aligned Block Copy Problem" on page 63 for another example.

Figure 3 Linked List Search Code in C

```
struct block {
        struct block *link;            /* link to next block */
        int pad1[11];
        int data;                      /* data item to check for */
        int pad2[12];
        } blocks[5000];

struct block *find(pb,value)
        struct block *pb;
        int value;
        {
        while(pb)                /* check for end of linked list */
                {
                if (pb->data == value) /* check for value match */
                        return pb;     /* return matching block */
                pb = pb->link;         /* follow link to next block */
                }
        return (struct block *)0; /* return null pointer if no match */
        }
```

The while loop compiles to just seven instructions, including two loads, two tests, two branches and a no-op. Note that the instruction after a branch is always executed on a SPARC processor.

Figure 4 Linked List Search Loop in Assembler

```
LY1:                            /* loop setup code omitted */
        cmp     %o2,%o1         /* see if data == value */
        be      L77016          /* and exit loop if matched */
        nop                     /* pad branch delay slot */
        ld      [%o0],%o0       /* follow link to next block */
        tst     %o0             /* check for end of linked list */
        bne,a   LY1             /* branch back to start of loop */
        ld      [%o0+48],%o2    /* load data during branch delay slot */
```

Cache Miss Cost And Hit Rates For Different Machines

Since the hardware details vary from one machine implementation to another and the details are sometimes hard to obtain, the cache architectures of some common machines are described below, divided into four main groups. Virtually and physically addressed caches with write back algorithms, virtual write through caches and on-chip caches. Discussion of multiprocessor cache issues is left to Chapter , “Multiprocessors”

Virtual Write Through Caches

Example Machines

Most older desktop SPARCstation’s from Sun and the deskside SPARCsystem 300 series use this cache type.

How It Works

The cache works using virtual addresses to decide which location in the cache has the required data in it. This avoids having to perform an MMU address translation except when there is a cache miss.

Data is read into the cache a block at a time but writes go through the cache into main memory as individual words. This avoids the problem of data in the cache being different from data in main memory but may be slower since single word writes are less efficient than block writes would be. An optimisation is for a buffer to be provided so that the word can be put into the buffer then the CPU can continue immediately while the buffer is written to memory. The depth (one or two words) and width (32 or 64 bits) of the write buffer vary. If a number of words are written back-to-back then the write buffer may fill up and the processor will stall until the slow memory cycle has completed. A doubleword write on a SPARCstation 1 (and similar machines) will always cause a write buffer overflow stall that takes 4 cycles.

On the machines tabled below the processor waits until the entire cache block has been loaded before continuing.

The SBus used for memory accesses on the SS2, IPX and ELC machines runs at half the CPU clock rate and this may give rise to an extra cycle on the miss cost to synchronize the two buses which will occur half of the time on average.

Table 10 Virtual Write Through Cache Details

Machine	Clock	Size	Line	Read Miss Cost	WB size	WB Full Cost
SS1, SLC	20MHz	64Kb	16 b	12 cycles	1 word	2 cycles (4 dbl)
SS1+, IPC	25MHz	64Kb	16 b	13 cycles	1 word	2 cycles (4 dbl)
SPARCsystem 300	25MHz	128Kb	16 b	18 cycles	1 double	2 cycles
ELC	33MHz	64Kb	32 b	24-25 cycles	2 doubles	4-5 cycles
SS2, IPX	40MHz	64Kb	32 b	24-25 cycles	2 doubles	4-5 cycles

Virtual Write Back Caches

Example Machines

The high end deskside SPARCserver's from Sun use this cache type.

How It Works

The cache uses virtual addresses as described above. The difference is that data written to the cache is not written through to main memory. This reduces memory traffic and allows efficient back-to-back writes to occur. The penalty is that a cache line must be written back to main memory before it can be reused so there may be an increase in the miss cost. The line is written efficiently as a block transfer, then the new line is loaded as a block transfer. Most systems have a buffer which is used to store the outgoing cache line while the incoming cache line is loaded, then the outgoing line is passed to memory while the CPU continues. The SPARCsystem 400 backplane is 64 bits wide and runs at 33MHz, synchronised with the CPU. The SPARCsystem 600 uses a 64 bit MBus and

Table 11 Virtual Write Back Cache Details

Machine	Size	Line	Miss Cost	CPU Clock Rate
Sun4/200 series	128Kb	16 bytes	7 cycles	16 MHz
SPARCserver 400	128Kb	32 bytes	12 cycles	33 MHz
SPARCserver 600 model 120	64Kb	32 bytes	28 cycles?	40 MHz

takes data from the MBus at full speed into a buffer but the cache itself is 32 bits wide and it takes extra cycles to pass data from the buffer in the cache controller to the cache itself.

Physical Write Back Caches

Example Machines

SPARC compatibles from Solbourne and ICL, and the SPARCserver 600 and SPARCstation 10 with 2nd level SuperCache use this cache type.

How It Works

The MMU translations occur on every CPU access before the address reaches the cache logic. The cache uses the physical address of the data in main memory to determine where in the cache it should be located. In other respects this type is the same as the Virtual Write Back Cache described above.

The KBus backplane on Solbourne systems and the HSPbus on ICL systems are 64 bits wide and run at about 16 MHz with a read throughput quoted by ICL of 66Mb/s. The Memory system on the SPARCstation 10 uses special SIMMs and it is faster than the SPARCserver 600MP by one cycle on a cache miss.

The SuperCache controller implements sub-blocking in its 1 Mb cache. The cache line is actually 128 bytes but it is loaded as four separate contiguous 32 byte lines. This quarters the number of cache tags required at the expense of needing an extra three valid bits in each tag. In XBus mode for the SPARCcenter 2000 the same chipset switches to 64 byte lines

Table 12 Physical Write Back Cache Details

Machine	Size	Line	Miss Cost	CPU Clock Rate
Solbourne S5	128Kb	32 bytes	18 cycles?	33 MHz
Solbourne S5E	128Kb	32 bytes	21 cycles?	40 MHz
ICL DRS6000/50	128Kb	32 bytes	18 cycles?	33 MHz
SPARCserver 600 model 41	1024Kb	128 (32) bytes	18 cycles?	40MHz
SPARCstation 10 model 41	1024Kb	128 (32) bytes	17 cycles?	40MHz
SPARCcenter 2000	1024Kb	128 (64) bytes	N/A	40MHz

On-Chip Caches

Example Machines

Highly integrated SPARC chipsets like the Matsushita MN10501 used in the Solbourne S4000 the Texas Instruments SuperSPARC (Viking) used in the SPARCstation 10 models 20 and 30, MicroSPARC (Tsunami) used in the SPARCstation LX and SPARCclassic and the Fujitsu MB86930 (SPARClite) use this cache type. The Cypress HyperSPARC uses a hybrid on-chip instruction cache with off chip unified cache.

How It Works

Since the entire cache is on-chip, complete with its control logic, a different set of trade-offs apply to cache design. The size of the cache is limited but the complexity of the cache control logic can be enhanced more easily. On-chip caches may be *associative* in that a line can exist in the cache in several possible locations. If there are four possible locations for a line then the cache is known as a *four way set associative cache.* It is hard to build off chip caches that are associative so they tend to be *direct mapped*, where each memory location maps directly to a single cache line.

On-Chip caches also tend to be split into separate instruction and data caches since this allows both caches to transfer during a single clock cycle which speeds up load and store instructions. This is not normally done with off-chip caches because the chip would need an extra set of pins and more chips on the circuit board.

More intelligent cache controllers can reduce the miss cost by passing the memory location that missed as the first word in the cache block, rather than starting with the first word of the cache line. The processor can then be allowed to continue as soon as this word arrives, before the rest of the cache line has been loaded. If the miss occurred in a data cache and the processor can continue to fetch instructions from a separate instruction cache then this will reduce the miss cost. On a combined instruction and data cache the cache load operation keeps the cache busy until it has finished so the processor cannot fetch another instruction anyway. SuperSPARC and MicroSPARC both implement this optimisation.

MicroSPARC uses page mode DRAM to reduce its miss cost. The first miss to a 1Kb region takes 9 cycles for a data miss, consecutive accesses to the same region avoid some DRAM setup time and complete in 4 cycles.

The SuperSPARC processor implements sub-blocking in its instruction cache. The cache line is actually 64 bytes but it is loaded as two separate contiguous 32 byte lines.

Table 13 On-Chip Cache Details

Processor	I-Size	I-line	I-Assoc	D-Size	D-line	D-Assoc	D-Miss Cost
MB86930	2Kb	16	2	2Kb	16	2	?
MN10501	6Kb	?	?	2Kb	?	?	?
SuperSPARC	20Kb	64 (32)	5	16Kb	32	4	12 cycles
MicroSPARC	4Kb	32	1	2 Kb	16	1	4-9 cycles
HyperSPARC	8Kb	32	1	256Kb[1]	32	1	?

1. This is a combined external instruction and data cache.

The SuperSPARC Two Level Cache Architecture

External Cache

As described above the SuperSPARC processor has two sophisticated and relatively large on-chip caches and an optional 1 Mbyte external cache[1]. It can be used without the external cache, and the on-chip caches work in write-back mode for transfers directly over the MBus. For multiprocessor snooping to work correctly and efficiently the on-chip caches work in write through mode when the external cache is used. This guarantees that the on-chip caches contain a subset of the external cache so that snooping is only required on the external cache. There is an 8 doubleword (64byte) write buffer that flushes through to the external cache.

1. "The SuperSPARC Microprocessor Technical White Paper"

Multiple Miss Handling

A very advanced feature of the external cache controller is that one read miss and one write miss can be handled at any given time. When a processor read access incurs a miss the controller can still allow the processor to access the external cache for writes (such as write buffer flushes) until a write miss occurs. Conversely when a write miss occurs the processor can continue to access the cache for reads (such as instruction prefetches) until a read miss occurs.

Asynchronous MBus Interface

The external cache controller provides an asynchronous interface to the MBus that allows the processor and both sets of caches to run at a higher clock rate than the MBus. This means that the miss cost is variable, depending on the presence or not of the external cache and the relative clock rate of the cache and MBus. The transfer times between the two caches are fixed however and a 32 byte transfer into an on-chip cache takes a total of 6 cycles. The data is transferred critical word first and the processor only suffers a 4 cycle miss cost. For writes there is a 3 cycle latency then 8 bytes are transferred on each cycle.

Efficient Register Window Overflow Handling

One common case in SPARC systems is a register window overflow trap. This involves 8 consecutive doubleword writes to save the registers. All 8 writes can fit in the write buffer and they can be written to the second level cache in two 32 byte bursts.

I/O Caches

If an I/O device is performing a DVMA transfer, e.g. a disk controller is writing data into memory, the CPU can continue other operations while the data is transferred. Care must be taken to ensure that the data written to by the I/O device is not also in the cache, otherwise inconsistencies can occur. On older Sun systems, and the 4/260 and SPARCsystem 300, every word of I/O is passed through the cache[1]. When a lot of I/O is happening this slows down the CPU since it cannot access the cache for a cycle. The SPARCsystem 400 has

1. “Sun Systems and their Caches, by Sales Tactical Engineering June 1990.”

an I/O cache which holds 128 lines of 32 bytes and checks its validity with the CPU cache once for each line. The interruption to the CPU is reduced from once every 4 bytes to once every 32 bytes. The other benefit is that single cycle VMEbus transfers are converted by the I/O cache into cache line sized block transfers to main memory which is much more efficient[1]. The SPARCserver 600 has a similar I/O cache on its VMEbus to SBus interface[2] but it has 1024 lines rather than 128. The SBus to MBus interface can use block transfers for all I/O so does not need an I/O cache but it does have its own I/O MMU and I/O is performed in a cache coherent manner on the MBus in the SPARCserver 10 and SPARCserver 600 machines.

Kernel Block Copy

The kernel spends a large proportion of its time copying or zeroing blocks of data. These may be internal buffers or data structures but a common operation involves zeroing or copying a page of memory, which is 4Kb or 8Kb. The data is not often used again immediately so it does not need to be cached. In fact the data being copied or zeroed will normally remove useful data from the cache. The standard C library routine for this is called “bcopy” and it handles arbitrary alignments and lengths of copies. If you know that the data is aligned and a multiple of the cache block size then a simpler and faster copy routine can be used.

Software Page Copies

The most efficient way to copy a page on a system with a write back cache is to read a cache line then write it as a block, using two bus transactions. The sequence of operations for a software copy loop is actually:

- load the first word of the source causing a cache miss
- fetch the entire cache line from the source
- write the first word to the destination causing a cache miss
- fetch the entire cache line from the destination (the system cannot tell that you are going to overwrite all the old values in the line)
- copy the rest of the source cache line to the destination cache line

1. “A Cached System Architecture Dedicated for the System IO Activity on a CPU Board, by Hseih, Wei and Loo.
2. “SPARCserver 10 and SPARCserver 600 White Paper”

- at some later stage the destination cache line will be written back to memory when the line is reused by another read
- go back to the first stage in the sequence, using the next cache line for the source and destination

The above sequence is fairly efficient but actually involves three bus transactions since the source data is read, the destination data is read unnecessarily and the destination data is written. There is also a delay between the bus transactions while the cache line is copied.

The Classic Cache Aligned Block Copy Problem

A well known cache-busting problem can occur with direct mapped caches when the buffers are aligned with the source and destination addresses an exact multiple of the cache size apart. Both the source and destination use the same cache line and a software loop doing 4 byte loads and stores with a 32 byte cache line would cause 8 read misses and 8 write misses for each cache line copied, instead of two read misses and one write miss. This is desperately inefficient and can be caused by simple coding.

```
#define BUFSIZE 0x10000/* 64Kbytes matches SS2 cache size */
char source[BUFSIZE], destination[BUFSIZE];
...
bcopy(source, destination, BUFSIZE);
```

The compiler will allocate both arrays adjacently in memory so they will be aligned and the bcopy will run very slowly.

Kernel Bcopy Acceleration Hardware

The SPARCserver 400 series machines and the SuperCache controller implement hardware bcopy acceleration. It is controlled by sending commands to special cache controller circuitry. The commands use privileged instructions (Address Space Identifier or ASI loads and stores) that cannot be used by normal programs but are used by the kernel to control the memory management unit and cache controller in all SPARC machines. The SPARCserver 400 and SuperCache have extra hardware that uses a special cache line buffer within the cache controller and use special ASI load and store

addresses to control the buffer. A single ASI load causes a complete line load into the buffer from memory and a single ASI write causes a write of the buffer to memory. The data never enters the main cache so none of the existing cached data is overwritten and the ideal pair of bus transactions occur back to back with minimal delays in-between and use all the available memory bandwidth. An extra ASI store is defined that writes values into the buffer so that block zero can be implemented by writing zero to the buffer and performing block writes of zero at full memory speed without also filling the cache with zeroes. Physical addresses are used so the kernel has to look up the virtual to physical address translation before it uses the ASI commands.The SuperCache controller is present in SPARCstation 10 and SPARCserver 600 machines that have the SuperSPARC with 1Mb external cache only.

Windows and Graphics

This subject is too big to address here but references are included to some existing papers that cover performance issues for the windows and graphics layer. One point to note is that the XGL product comes with a collection of benchmark demo programs which have many parameters that can be set interactively to determine the performance of a system under your own conditions.

- "OpenWindows Version 2 White Papers"
- "SunPHIGS / SunGKS Technical White Paper"
- "XGL Graphics Library Technical White Paper"
- "Graphics Performance Technical White Paper, January 1990"
- "SPARCserver and SPARCcenter Performance Brief"

Disk

Disk usage can be tuned to some extent, but understanding the effect that a different type of disk or controller may make to your system is an important part of performance tuning. Hopefully this chapter will enable you to understand how to interpret the specifications often quoted for disks, and to work out whether you are getting the throughput you should be!

The Disk Tuning Performed For SunOS 4.1.1

As ever, more recent versions of SunOS are the best tuned. In particular the UFS file system code was extensively tuned for SunOS 4.1.1 with the introduction of an I/O clustering algorithm, which groups successive reads or writes into a single large command to transfer up to 56Kb rather than lots of 8Kb transfers. The change allows the filesystem layout to be tuned to avoid sector interleaving and allows filesystem I/O on sequential files to get close to its theoretical maximum[1].

If a disk is moved from a machine running an earlier release of SunOS to one running SunOS 4.1.1 then its sectors will be interleaved and the full benefit will not be realised. It is advisable to backup the disk, rerun newfs on it and restore the data[2].

Throughput Of Various Common Disks

Understanding the Specification

It is said that there are “lies, damned lies, and disk specifications...”. Lets try to make some sense of them.

1. “Extent-like Performance from a Unix File System, L. McVoy and S. Kleiman”

2. See also the manual page for tunefs(8).

What the Makers Specify

The disk manufacturers specify certain parameters for a drive. These can be misinterpreted since they are sometimes better than you can get in practice.

- Rotational speed in revolutions per minute (rpm)
- The number of tracks or cylinders on the disk
- The number of heads or surfaces in each cylinder
- The rate at which data is read and written (MHz or Millions of bytes/s)
- The disk controller interface used (ST506, ESDI, SCSI, SMD, IPI)
- The *unformatted capacity* of the drive (Millions of bytes)
- The average and single track *seek time* of the disk

What the System Vendors Specify

The system vendors need to deal with the disk in terms of sectors, typically containing 512 bytes of data each and many bytes of header, preamble and inter-sector gap each. Spare sectors and spare cylinders are also allocated so that bad sectors can be substituted. This reduces the unformatted capacity to what is known as the *formatted capacity.* For example a 760Mb drive reduces to a 669Mb drive when the format is taken into account. The `format` command is used to write the sectors to the disk. The file `/etc/format.dat` contains information about each type of disk and how it should be formatted. The *formatted* capacity of the drive is measured in 10^6 Mbytes (1000000) while RAM sizes are measured in 2^{20} Mbytes (1048576). Confused? You will be!

What You have to Work Out for Yourself

You can work out, using information from `/etc/format.dat`, the real peak throughput and size in Kbytes (1024) of your disk. The entry for a typical disk is shown in figure 8-1.

Figure 1 /etc/format.dat entry for Sun 669Mb disk

```
disk_type = "SUN0669" \
        : ctlr = MD21 : fmt_time = 4 \
        : trks_zone = 15 : asect = 5 : atrks = 30 \
        : ncyl =  1614 : acyl = 2 : pcyl = 1632 : nhead = 15 : nsect = 54 \
        : rpm = 3600 : bpt = 31410
```

The values to note are:

- rpm = 3600, so the disk spins at 3600 rpm
- nsect = 54, so there are 54 sectors of 512 bytes per track
- nhead = 15, so there are 15 tracks per cylinder
- ncyl = 1614, so there are 1614 cylinders per disk

Since we know that there are 512 bytes per sector, 54 sectors per track and that a track will pass by the head 3600 times per minute we can work out the peak sustained data rate and size of the disk.

data rate (bytes/sec) = (nsect * 512 * rpm) / 60 = 1658880 bytes/sec

size (bytes) = nsect * 512 * nhead * ncyl = 669358080 bytes

If we assume that 1 Kbyte is 1024 bytes then the data rate is 1620 Kbytes/sec.

The manufacturer (and Sun) rate this disk at 1.8 Mbytes/s, which is the data rate during a single sector. This is in line with industry practice, but it is impossible to get better than 1620 Kbytes/sec for the typical transfer size of between 2 and 56Kb. Sequential reads on this type of disk tend to run at just over 1500 Kbytes/sec which confirms the calculated result. Some common Sun disks are listed in table 8-1 with Kbytes of 2^{10} = 1024. The calculated data rate for ZBR drives is the peak value as it varies across the cylinders.

Table 14 Disk Specifications

Disk Type	Calculated Capacity	Peak	Data Rate	RPM	Seek
Quantum 105S	102270 Kb	1.2	1068 Kb/s	3662	20ms
Sun 0207 SCSI	203148 Kb	1.6	1080 Kb/s	3600	16ms
Sun 0424 ZBR SCSI	414360 Kb	2.5-3.0	2933 Kb/s	4400	14ms
Sun 0669 SCSI	653670 Kb	1.8	1620 Kb/s	3600	16ms
Sun 1.3G ZBR SCSI	1336200 Kb	3.25-4.5	3600 Kb/s	5400	11ms
Sun 1.3G ZBR IPI	1255059 Kb	3.25-4.5	3510 Kb/s	5400	11ms
Hitachi 892M SMD	871838 Kb	2.4	2010 Kb/s	3600	15ms
Sun 1.05G ZBR FSCSI	1026144 Kb	2.9-5.1	3240 Kb/s	5400	11ms
CDC 911M IPI	889980 Kb	6.0	4680 Kb/s	3600	15ms
Sun 2.1G ZBR DFSCSI	20770800 Kb	3.8-5.0	3600 Kb/s	5400	11ms

ZBR Drives

These drives vary depending upon which cylinder is accessed. The disk is divided into zones with different bit rates (ZBR) and the outer part of the drive is faster and has more sectors per track than the inner part of the drive. This allows the data to be recorded with a constant linear density along the track (bits per inch). In other drives the peak number of bits per inch that can be made to work reliably is set up for the innermost track but density is too low on the outermost track. In a ZBR drive more data is stored on the outer tracks so greater capacity is possible. The 1.3Gb drive zones mean that peak performance is obtained from the first third of the disk up to cylinder 700.

Table 15 1.3Gb IPI ZBR Disk Zone Map

Zone	Start Cylinder	Sectors per Track	Data Rate in Kbytes/s
0	0	78	3510
1	626	78	3510
2	701	76	3420
3	801	74	3330
4	926	72	3240
5	1051	72	3240
6	1176	70	3150
7	1301	68	3060
8	1401	66	2970
9	1501	64	2880
10	1601	62	2790
11	1801	60	2700
12	1901	58	2610
13	2001	58	2610

Note that the format.dat entry assumes constant geometry so it has a fixed idea about sectors per track and the number of cylinders in format.dat is reduced to compensate. It is not clear whether the variable geometry causes any performance problems due to optimisations in the filesystem and disk driver code that may assume fixed geometry.

Sequential Versus Random Access

Some people are surprised when they read that a disk is capable of several megabytes per second but they see a disk at 100% capacity providing only a few hundred kilobytes per second for their application. Most disks used on NFS or database servers spend their time serving the needs of many users and the access patterns are essentially random. The time taken to service a disk access is taken up by seeking to the correct cylinder and waiting for the disk to go round. In sequential access the disk can be read at full speed for a complete cylinder but in random access the average seek time quoted for the disk should be allowed for between each disk access. The random data rate is thus very dependent on how much data is read on each random access. For filesystems 8Kbytes is a common block size but for databases on raw disk partitions 2Kbytes is a common block size. The 1.3Gb disk takes 11ms for a random seek and takes about 0.5ms for a 2Kb transfer and 2ms for an 8Kb transfer. The data rate is thus:

data rate = transfersize / (seektime + (transfersize / datarate))

2Kb data rate = 173 = 2 / (0.011 + (2 / 3510))

8Kb data rate = 602 = 8 / (0.011 + (8 / 3510))

56Kb data rate = 2078 = 56 / (0.011 + (56 / 3510))

Anything that can be done to turn random access into sequential access or to increase the transfer size will have a significant effect on the performance of a system. This is one of the most profitable areas for performance tuning.

Effect Of Disk Controller, SCSI, SMD, IPI

SCSI Controllers

SCSI Controllers can be divided into four generic classes. The oldest only support *Asynchronous SCSI,* more recent controllers support *Synchronous SCSI,* the latest support *Fast Synchronous SCSI,* and *Differential Fast Synchronous SCSI.* Working out which type you have on your Sun is not that simple. The main differences between them is in the number of disks that can be supported on a SCSI bus before the bus becomes saturated and the maximum effective cable length allowed.

Fast SCSI increases the maximum data rate from 5 Mbytes/s to 10 Mbytes/s and halves the cable length from 6 meters to 3 meters. Differential Fast SCSI increases the cable length to 25 meters but uses incompatible electrical signals and a different connector so can only be used with devices that are purpose built.

Most of the SCSI disks shown above transfer data at a much slower rate. They do however have a buffer built into the SCSI drive which collects data at the slower rate and can then pass data over the bus at a higher rate. With Asynchronous SCSI the data is transferred using a handshake protocol which slows down as the SCSI bus gets longer. For fast devices on a very short bus it can achieve full speed but as devices are added and the bus length and capacitance increases the transfers slow down. For Synchronous SCSI the devices on the bus negotiate a transfer rate which will slow down if the bus is long but by avoiding the need to send handshakes more data can be sent in its place and throughput is less dependent on the bus length. The transfer rate is printed out by the device driver as a system boots[1] and is usually 3.5 to 5.0 Mb/s but could be up to 10.0 Mb/s for a fast SCSI device on a fast SCSI controller.

The original SPARCstation 1 and the VME hosted SCSI controller used by the SPARCserver 470 do not support Synchronous SCSI. In the case of the SPARCstation 1 it is due to SCSI bus noise problems that were solved for the SPARCstation 1+ and subsequent machines. If noise occurs during a Synchronous SCSI transfer a SCSI reset happens and, while disks will retry, tapes will abort. In versions of SunOS before SunOS 4.1.1 the SPARCstation 1+, IPC, SLC have Synchronous SCSI disabled. The VME SCSI controller supports a maximum of 1.2Mbytes/s while the original SBus SCSI supports 2.5 Mbytes/s because it shares its DMA controller bandwidth with the ethernet.

The SPARCstation 2, IPX and ELC use a higher performance SBus DMA chip than the SPARCstation 1, 1+, SLC and IPC and they can drive sustained SCSI bus transfers at 5.0 Mb/s. The SBus SCSI add-on cards[2] and the SPARCstation 330 are also capable of this speed.

The SPARCstation 10 introduced the first fast SCSI implementation from Sun, together with a combined fast SCSI/Ethernet SBus card. A 1.0Gbyte 3.5” fast SCSI drive with similar performance to the older 1.3Gb drive was also

1. WIth SunOS 4.1.1 and subsequent releases.

2. The X1055 SCSI controller and the X1054 SBE/S SCSI/Ethernet card.

announced in the high end models. More recently a differential version of the fast SCSI controller has been introduced, together with a 2.1Gb differential fast SCSI drive that has a new tagged command queueing interface on-board. TCQ provides optimisations similar to those implemented on IPI controllers described later in this chapter, but the buffer and optimisation occurs in each drive rather than in the controller for a string of disks.

SMD Controllers

There are two generations of SMD controllers used by Sun. The Xylogics 451 (xy) controller is a major bottleneck since it is saturated by a single slow disk and you are doing well to get better than 500Kb/s through it. They should be upgraded if at all possible to the more recent SMD-4 (Xylogics 7053, xd) controller, which will provide several times the throughput. SMD Disks are no longer sold by Sun, but the later model 688 and 892 Mb disks provided peak bandwidth of 2.4 Mb/s and a real throughput of around 1.8Mb/s. The benefits of upgrading a xy451 with an 892 Mb drive are obvious!

IPI Controllers

The I/O performance of a multi-user system depends upon how well it supports randomly distributed disk I/O accesses. The ISP-80 disk controller was developed by Sun Microsystems to address this need. It provides much higher performance than SMD based disk subsystems. The key to its performance is the inclusion of a 68020 based real time seek optimiser which is fed with rotational position sensing (RPS) information so that it knows which sector is under the drives head. It has a 1 Mbyte disk cache organised as 128 - 8 Kbyte disc block buffers and can queue up to 128 read/write requests. As requests are added to the queue the requests are reordered into the most efficient disk access sequence. The result is that when a heavy load is placed on the system and a large number of requests are queued the performance is much better than competing systems. When the system is lightly loaded the controller performs speculative pre-fetching of disk blocks to try and anticipate future requests. This controller turns random accesses into sequential accesses and significantly reduces the average seek time for a disk by seeking to the nearest piece of data in its command queue rather than performing the next command immediately.

Load Monitoring And Balancing

If a system is under a heavy I/O load then the load should be spread across as many disks and disk controllers as possible. To see what the load looks like the `iostat` command can be used. This produces output in two forms, one looks at the total throughput of each disk in Kb/s and the average seek time and number of transfers per second, the other form looks at the number of read and write transfers separately and gives a percentage utilisation for each disk. Both forms also show the amount of terminal I/O and the CPU loading.

```
% iostat 5
      tty           sd0           sd1           sd3           cpu
 tin tout bps tps msps  bps tps msps  bps tps msps  us ni sy id
   0    0   1   0  0.0    0   0  0.0    0   0  0.0  15 39 23 23
   0   13  17   2  0.0    0   0  0.0    0   0  0.0   4  0  6 90
   0   13 128  24  0.0    0   0  0.0   13   3  0.0   7  0  9 84
   0   13  88  18  0.0    0   0  0.0   48  10  0.0  11  0 11 78
   0   13 131  24  0.0    0   0  0.0   15   3  0.0  21  0 20 59
   0   13  85  17  0.0    0   0  0.0   13   3  0.0  17  0 11 72
   0   13   0   0  0.0    0   0  0.0    0   0  0.0   6  0  4 90
   0   13  57   9  0.0    0   0  0.0   14   2  0.0  11  0  6 84
   0   13 108  18  0.0    0   0  0.0    9   2  0.0  39  0  7 54
```

```
 % iostat -tDc 5
      tty           sd0           sd1           sd3           cpu
 tin tout rps wps util  rps wps util  rps wps util  us ni sy id
   0    0   0   0  0.6    0   0  0.0    0   0  0.1  15 39 23 23
   0   13   0   2  4.9    0   0  0.0    0   0  0.0  12  0  8 79
   0   13   1   0  4.0    0   0  0.0    0   0  0.0  35  0 11 54
   0   13   7   0 23.1    0   0  0.0    2   0  7.6  21  0 11 68
   0   13   6   4 33.5    0   0  0.0    2   1  7.7   6  0  6 89
   0   13  11   1 32.3    0   0  0.0    0   0  0.8  12  0 11 77
   0   13  32   1 83.9    0   0  0.0    0   0  0.0  54  0 17 29
   0   13   3   0 11.0    0   0  0.0    0   0  0.0  13  0  5 82
   0   13   2   1  8.7    0   0  0.0    0   0  0.0  23  0 14 63
```

The second form of `iostat` was introduced in SunOS 4.1. The above output shows paging activity on a SPARCstation 1 with Quantum 104Mb disks, the SCSI device driver does not collect seek information so the `msps` field is always zero. It can be used with SMD and IPI disks to see whether the disk

activity is largely sequential (less than the rated seek time) or largely random (equal or greater than the rated seek time). The first line of output shows the average activity since the last reboot, like `vmstat`.

The %util field is the most useful. The device driver issues commands to the drive and measures how long it takes to respond and how much time the disk is idle between commands to produce a %util figure. A 100% busy disk is one that has no idle time before the next command is issued. Disks that peak at over 50% busy during a 5 second interval are probably causing performance problems.

Multiple Disks On One Controller

If the controller is busy talking to one disk then another disk has to wait and the resulting contention increases the latency for all the disks on that controller. Empirical tests have shown that with the current 424Mb and 1.3Gb disks on a 5Mb/s SCSI bus there should be at most two heavily loaded disks per bus for peak performance. IPI controllers can support at most three heavily loaded disks. As the data rate of one disk approaches the bus bandwidth the number of disks is reduced. The 10Mb/s fast SCSI buses with disks that transfer over the bus at 10Mb/s and sustain about 4Mb/s data rate will be able to support about four heavily loaded disks per bus.

Mirrored Disks

Sun's Online: DiskSuite product includes a software driver for disk mirroring. When a write occurs the data must be written to both disks together and this can cause a bottleneck for I/O. The two mirrored disks should be on completely separate IPI disk controllers or SCSI buses so that the write can proceed in parallel. If this advice is not followed the writes happen in sequence and the disk write throughput is halved. Putting the drives on separate controllers will also improve the system's availability since disk controller and cabling failures will not prevent access to both disks.

Tuning Options For Mirrors

Filesystem Tuning Fix

The Online: DiskSuite product was developed at about the time filesystem clustering[1] was implemented and it doesn't set up the tuning parameters correctly for best performance. After creating a filesystem but before mounting it you should run the following command on each metapartition.

```
# tunefs -a 7 -d 0 /dev/mdXX
```

The -a option controls the `maxcontig` parameter. This specifies the maximum number of blocks, belonging to the same file, that will be allocated contiguously before inserting a rotational delay.

The -d option controls the `rotdelay` parameter. This specifies the expected time (in milliseconds) to service a transfer completion interrupt and initiate a new transfer on the same disk. It is used to decide how much rotational spacing to place between successive blocks in a file. For drives with track buffers a rotdelay of 0 is usually the best choice.

Metadisk Options

There are several options that can be used in the metadisk configuration file to control how mirroring operates.

Writes always go to both disks so the only option is whether to issue two write requests then wait for them both to finish or to issue them sequentially. The default is simultaneous issue which is what is wanted in most cases.

Reads can come from either disk and there are four options. The first is to issue reads to each disk simultaneously and when one completes to cancel the second one, this aims to provide the fastest response but generates extra work and will not work well in a heavily loaded system. Reads can be made alternately from one disk then the other, balancing the load but not taking advantage of any sequential prefetching that disks may do. Reads can be made to come from one disk only (except on failure), which can be useful if there are two separate read intensive mirrored partitions sharing the same disk. Each partition can be read from a different disk. The last option is to geometrically

1. See "The Disk Tuning Performed For SunOS 4.1.1" on page 67, and the tunefs manual page.

divide the partition into two and to use the first disk to service reads from the first half of the partition and the second disk to service reads from the second half. For a read intensive load the heads will only have to seek half as far so this reduces the effective average seek time and provides better performance for random accesses.

CPU

This chapter looks at some SPARC implementations to see how the differences affect performance. A lot of hard-to-find details are documented.

Architecture and Implementation

The SPARC architecture defines everything that is required to ensure application level portability across varying SPARC implementations. It intentionally avoids defining some things, like how many cycles an instruction takes, to allow maximum freedom within the architecture to vary implementation details. This is the basis of SPARC's scalability from very low cost to very high performance systems. Implementation differences are handled by kernel code only, so that the instruction set and system call interface are the same on all SPARC systems. The SPARC Compliance Definition Version 1.0 defines this interface for SunOS 4.X. Within this standard there is no specification of the performance of a compliant system, only its correctness. The performance depends on the chip set used (i.e. the implementation) and the clock rate that the chip set runs at. To avoid confusion some terms need to be defined.

Instruction Set Architecture (ISA)

Defined by the SPARC Architecture Manual, Sun has published Version 7 and Version 8, the IEEE have a draft standard based on Version 8, IEEE P1754. Version 9 is just emerging and defines major extensions including 64 bit addressing in an upwards compatible manner for user mode programs.

SPARC Implementation

A chip level specification, it will define how many cycles each instruction takes and other details. Some chip sets only define the integer unit (IU) and floating point unit (FPU), others define the MMU and cache design and may include the whole lot on a single chip.

System Architecture

This is more of a board level specification of everything the kernel has to deal with on a particular machine. It includes internal and I/O bus types, address space uses and built-in I/O functions. This level of information is documented in the SPARCengine User Guide[1] that is produced for the bare board versions of Sun's workstation products. The information needed to port a real-time operating system to the board and physically mount the board for an embedded application is provided.

Kernel Architecture

A number of similar systems may be parameterised so that a single GENERIC kernel image can be run on them. This grouping is known as a kernel architecture and Sun has one for VME based SPARC machines (Sun4), one for SBus based SPARC machines (Sun4c), one for the VME and SBus combined 6U eurocard machines (sun4e), one for MBus based machines and machines that use the SPARC reference MMU (Sun4m), and one for XDbus based machines (Sun4d), these are listed in Table 16 on page 83.

The Effect Of Register Windows And Different SPARC CPUs

SPARC defines an instruction set that uses 32 integer registers in a conventional way but has many sets of these registers arranged in overlapping *register windows*. A single instruction is used to switch in a new set of registers very quickly. The overlap means that 8 of the registers from the previous window are part of the new window, and these are used for fast parameter passing between subroutines. A further 8 global registers are always the same so, of the 24 that make up a register window, 8 are passed in from the previous window, 8 are local and 8 will be passed out to the next window. This is described further in the following references.

- "The SPARC Architecture Manual Version 8"
- "SPARC Strategy and Technology, March 1991"

1. See the "SPARCengine IPX User Guide", "SPARCengine 2 User Guide" and "SPARCengine 10 User Guide".

Some SPARC implementations have 7 overlapping sets of register windows and some have 8. One window is always reserved for taking traps or interrupts, since these will need a new set of registers, the others can be thought of as a stack cache for 6 or 7 levels of procedure calls with up to 6 parameters per call passed in registers. The other two registers are used to hold the return address and the old stack frame pointer. If there are more than 6 parameters to a call then the extra ones are passed on the external stack as in a conventional architecture. It can be seen that the register windows architecture allows much faster subroutine call and return and faster interrupt handling than conventional architectures which copy parameters out to a stack, make the call, then copy the parameters back into registers. Programs typically spend most of their time calling up and down a few levels of subroutines but when the register windows have all been used a special trap takes place and one window (16 registers) is copied to the stack in main memory. On average register windows seem to cut down the number of loads and stores required by 10-30% and provide a speed up of 5-15%. Care must be taken to avoid writing code that makes a large number of recursive or deeply nested calls, and keeps returning to the top level. If very little work is done at each level and very few parameters are being passed the program may generate a large number of save and restore traps. The SPARC optimiser can perform tail recursion elimination and leaf routine optimisation to reduce the depth of the calls.

If an application performs a certain number of procedure calls and causes a certain number window traps the benefit of reducing the number of loads and stores must be balanced against the cost of the traps. The overflow trap cost is very dependent upon the time taken to store 8 double-words to memory. On systems with write through caches and small write buffers like the SPARCstation 1 a large number of write stalls occur and the cost is relatively high. The SPARCstation 2 has a larger write buffer of two double-words which is still not enough. The SuperSPARC chip in write through mode has an 8 double-word write buffer so will not stall and systems with write back caches will not stall (unless a cache line needs to be updated)[1].

1. See Section , “Cache Miss Cost And Hit Rates For Different Machines,” on page 56.

The Effect Of Context Switches And Interrupts

When a program is running on a SPARC chip the register windows act as a stack cache and provide a performance boost. Subroutine calls tend to occur every few microseconds on average in integer code but may be infrequent in vectorisable floating point code. Whenever a context switch occurs the register windows are flushed to memory and the stack cache starts again in the new context. Context switches tend to occur every few milliseconds on average and a ratio of several hundred subroutine calls per context switch is a good one since there is time to take advantage of the register windows before they are flushed again. When the new context starts up it loads in the register windows one at a time, so programs that do not make many subroutine calls do not load registers that they will not need. Note that a special trap is provided that can be called to flush the register windows, this is needed if you wish to switch to a different stack as part of a user written co-routine or threads library. When running SunOS a context switch rate of 1000/second is considered fast so there are rarely any problems. There may be more concern about this ratio when running real time operating systems on SPARC machines, but there are alternative ways of configuring the register windows that are more suitable for real time systems[1]. These systems often run entirely in kernel mode and can perform special tricks to control the register windows.

The register window context switch time is a small fraction of the total SunOS context switch time. On machines with virtual write-back caches a cache flush is also required on a context switch. Systems have varying amounts of support for fast cache flush in hardware. The original SunOS 4.0 release mapped the U-area at the same address for all processes and the U-area flush gave the Sun4/260 with SunOS 4.0 (the first SPARC machine) a bad reputation for poor context switch performance that was mistakenly blamed on the register windows by some people.

Comparing Instruction Cycle Times On Different SPARC CPUs

Most SPARC Instructions execute in a single cycle. The main exceptions are floating point operations, loads, stores and a few specialised instructions. The time taken to complete each floating point operation in cycles is shown in Table 17 on page 86.

1. “SPARC Technology Conference notes - 3 Intense Days of Sun” - Alternative Register Window Schemes. The Alewife Project at MIT has implemented one of these schemes for fast context switching.

One factor that is not apparent from the table is that these devices have a floating point instruction queue, that is they can start new instructions before the old ones have finished. The queue is two to four instructions deep.

Table 16 Which SPARC IU and FPU does your system have?

System (Kernel Architecture)	Clock	Integer Unit	Floating Point Unit
Sun4/110 and Sun4/150 (sun4)	14MHz	Fujitsu #1	FPC+Weitek 1164/5
Sun4/260 and Sun4/280 (sun4)	16.6MHz	Fujitsu #1	FPC+Weitek 1164/5
Sun4/260 and Sun4/280 FPU2 (sun4)	16.6MHz	Fujitsu #1	FPC2+TI 8847
SPARCsystem 300 series (sun4)	25MHz	Cypress 601	FPC2+TI 8847
SPARCserver 400 series (sun4)	33MHz	Cypress 601	FPC2+TI 8847
SPARCstation 1 and SLC (sun4c)	20MHz	LSI/Fujisu #2	Weitek 3170
SPARCstation 1+ and IPC (sun4c)	25MHz	LSI/Fujitsu#2	Weitek 3170
Tadpole SPARCbook 1 (sun4m)	25MHz	Cypress 601	Weitek 3171
SPARCstation ELC (sun4c)	33MHz	Fujitsu #3	Weitek 3171 on chip
SPARCstation IPX (sun4c)	40MHz	Fujitsu #3	Weitek 3171 on chip
SPARCstation 2 (sun4c)	40MHz	Cypress 601	TI 602
SPARCserver 600 model 120/140 (sun4m)	40MHz	Cypress 601	Weitek 3171
SPARCsystem 10 model 20 (sun4m)	33MHz	SuperSPARC	SuperSPARC
SPARCsystem 10 model 30 (sun4m)	36MHz	SuperSPARC	SuperSPARC
SPARCsystem 10 & 600 model 41 (sun4m)	40MHz	SuperSPARC	SuperSPARC
SPARCsystem 10 & 600 model 52 (sun4m)	45MHz	SuperSPARC	SuperSPARC
SPARCclassic & SPARCstation LX (sun4m)	50MHz	MicroSPARC	MicroSPARC
SPARCcenter 2000 (sun4d)	40MHz	SuperSPARC	SuperSPARC
Cray S-MP & FPS 500EA (sun4?)	66MHz	BIT B5000	BIT B5010

Sun engineering teams designed all the above integer units which are listed by the vendor that manufactured and sold the parts. Sun has often dual sourced designs and several designs have been cross licenced by several vendors. The basic designs listed in Table 16 are:

- The original gate array IU design sold by Fujitsu (#1) as the MB86901 used a Weitek 1164 and 1165 based FPU initially with a floating point controller (FPC) ASIC but later moved to the Texas Instruments 8847 with a new FPC design.
- A tidied up semi-custom gate array version for higher clock rates is sold by Fujitsu and LSI Logic (#2) as the L64911. It takes two cycles to issue operations to the companion single chip Weitek 3170 FPU.
- A full-custom CMOS version with minor improvements and higher clock rates sold by Cypress as the CY7C601 which was licenced by Fujitsu (#3) and combined with a Weitek 3171 FPU into a single chip sold by both Fujitsu and Weitek as the MB86903. FPU operations are issued in one cycle to either a Texas Instruments 602 or a Weitek 3171 FPU.
- A high clock rate ECL implementation with an improved pipeline sold by BIT as the B5000. It has its own matching FPU design based on a custom ECL FPC and a standard BIT multiplier and accumulator.
- A second generation superscalar BICMOS implementation known as SuperSPARC with its own on-chip FPU sold by Texas Instruments as the TMS390Z50. The companion SuperCache controller chip is the TMS390Z55.
- A highly integrated, very low cost custom CMOS implementation known as MicroSPARC with an on-chip FPU derived from a design by Meiko Ltd. is sold by Texas Instruments as the TMS390S10.

There are several SPARC chips designed independently of Sun, including the Fujitsu MB86930 SPARClite embedded controller, the Matsushita MN10501 used by Solbourne and the Cypress HyperSPARC.

Superscalar Operations

The main superscalar SPARC chips are the Texas Instruments SuperSPARC and the Cypress/Ross HyperSPARC. These can both issue multiple instructions in a single cycle.

SuperSPARC

The SuperSPARC can issue three instructions in one clock cycle and just about the only instructions that cannot be issued continuously are integer multiply and divide, and floating point divide and square root as shown in Table 17.

There are a set of rules that control how many instructions are grouped for issue in each cycle. The main ones are that instructions are executed strictly in order and subject to the following conditions[1]:

- three instructions in a group
- one load or store anywhere in a group
- a control transfer (branch or call) ends a group at that point
- one floating point operation anywhere in a group
- two integer word results or one double word result (inc. loads) per group
- one shift can cascade into another operation but not vice versa

Dependant compare and branch is allowed and simple ALU cascades are allowed (a + b + c). Floating point load and a dependant operation is allowed but dependant integer operations have to wait for the next cycle after a load.

HyperSPARC

The HyperSPARC design can issue two instructions in one clock cycle. The combinations allowed are more restrictive than SuperSPARC but the simpler design allows a higher clock rate to compensate. I do not have comparative performance details for this chip and it is reported to be running in the lab but not shipping in product at present.

Low Cost Implementations

The main low cost SPARC chips are the Fujitsu SPARClite, a recent development aimed at the embedded control marketplace; and the Texas Instruments MicroSPARC, a highly integrated “workstation on a chip”. The MicroSPARC integer unit is based on the old BIT B5000 pipeline design although it has a new low cost FPU based upon a design by Meiko. The chip integrates IU, FPU, caches, MMU, SBus interface and a direct DRAM memory interface controller.

1. See “The SuperSPARC Microprocessor Technical White Paper”

Floating Point Performance Comparisons

The following table indicates why some programs that use divide or square root intensively may run better on a SPARCstation 2 than a SPARCstation IPX for example. The old Weitek 1164/5 did not implement square root in hardware and there were bugs in some early versions. The SunPro code generator avoids the bugs and the square root instruction in -cg87 mode but goes faster in -cg89 (optimised for the SPARCstation 2) or -cg92 (optimised for the SuperSPARC) mode if you know you do not have to run on the oldest type of machine. Solaris 2 disables the Weitek 1164/5 CPU so that -cg89 can be used as the default but this means that some old machines will revert to software FPU emulation. SunPro provide a command called “fpversion” that reports what you have in your machine.MicroSPARC has an interative FPU that is data dependent and minimum, typical and maximum times are given.

Table 17 Floating point Cycles per Instruction

Instruction	FPC & TI 8847	Weitek 3170 & 3171	Texas 602	BIT B5000	MicroSPARC min	MicroSPARC typ	MicroSPARC max	Super SPARC
fitos	8	10	4	2	5	6	13	1
fitod	8	5	4	2	4	6	13	1
fstoir, fstoi	8	5	4	2	6	6	13	1
fdtoir, fdtoi	8	5	4	2	7	7	14	1
fstod	8	5	4	2	2	2	14	1
fdtos	8	5	4	2	3	3	16	1
fmovs	8	3	4	2	2	2	2	1
fnegs	8	3	4	2	2	2	2	1
fabss	8	3	4	2	2	2	2	1
fsqrts	15	60	22	24	6	37	51	6
fsqrtd	22	118	32	45	6	65	80	10
fadds, fsubs	8	5	4	2	4	4	17	1
faddd, fsubd	8	5	4	2	4	4	17	1
fmuls	8	5	4	3	5	5	25	1
fmuld	9	8	6	4	7	9	32	1
fdivs	13	38	16	14	6	20	38	4
fdivd	18	66	26	24	6	35	56	7
fcmps, fcmpes	8	3	4	2	4	4	15	1
fcmpd, fcmped	8	3	4	2	4	4	15	1

Integer Performance Comparisons

Table 18 Number of Register Windows and Integer Cycles per Instruction

Instruction	Fujitsu/LSI #1, #2	Cypress/ Fujitsu #3	MicroSPARC (B5000 similar)	Fujitsu SPARClite	Super SPARC
(register windows)	7	8	7	8	8
ld (32 bit integer)	2	2	1	1	1
ldd (64 bit integer)	3	3	2	2	1
ld (32 bit float)	2	2	1	1	1
ldd (64 bit double)	3	3	2	2	1
st (32 bit integer)	3	3	2	1	1
std (64 bit integer)	4	4	3	2	1
st (32 bit float)	3	3	2	1	1
std (64 bit double)	4	4	3	2	1
taken branches	1	1	1	1	1
untaken branches	2	1	1	1	1
jmpl and rett	2	2	2	2	1
integer multiply	N/A	N/A	19	?	4
integer divide	N/A	N/A	39	?	18
issue FP operation	2	1	1	N/A	1

The main points of note in this table are that MicroSPARC is similar to the B5000 since they share the same basic pipeline design but FP loads and stores are faster on the B5000, and that SuperSPARC can issue up to three instructions in a clock cycle according to grouping rules mentioned previously. SPARClite does not include an FPU.

Multiprocessors

Multiprocessor machines introduce yet another dimension to performance tuning. Sun has been shipping multiprocessor servers for some time but the introduction of Solaris 2.1 and the desktop multiprocessor SPARCstation 10 is expected to bring multiprocessing into the mainstream both for end users and application developers. This chapter provides a brief description of how multiprocessor machines work and explains how to measure the utilisation of multiple processors to see if the existing processors are working effectively and to see if adding more processors would provide a worthwhile improvement.

Basic Multiprocessor Theory

Why Bother With More Than One CPU?

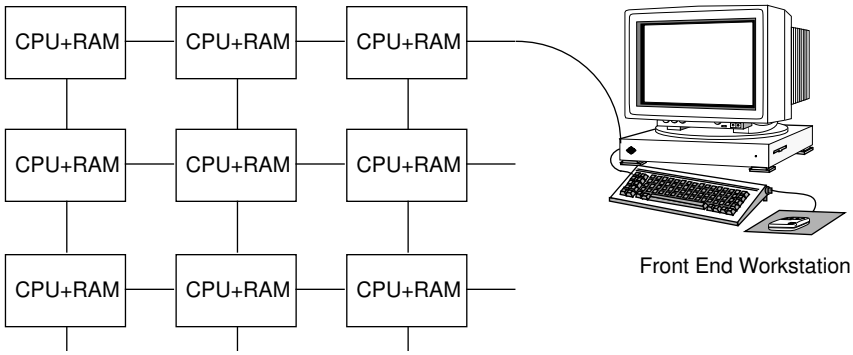
At any point in time there is are CPU designs that represent the best performance that can be obtained with current technology at a reasonable price. The cost and technical difficulty of pushing the technology further means that the most cost effective way of increasing computer power is to use several processors. There have been very many attempts to harness multiple CPUs and, today, there are many different machines on the market. Software has been the problem for these machines. It is almost impossible to design software that will be portable across a large range of machines and few of these machines sell in large numbers so there is a small and fragmented market for multiprocessor software.

Multiprocessor Classifications

There are two classes of multiprocessor machines that have some possibility of software compatibility and both have had SPARC based implementations.

Distributed Memory Multiprocessors

Figure 1 Typical Distributed Memory Multiprocessor With Mesh Network

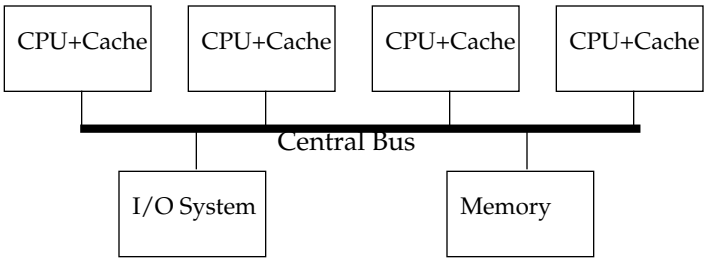


These can be though of as a network of uniprocessors packaged into a box. Each processor has its own memory and data must be explicitly copied over the network to another processor before it can be used. The benefit of this is that there is no contention for memory bandwidth to limit the number of processors and if the network is made up of point to point links the network throughput increases as the number of processor increases. There is no theoretical limit to the number of processors that can be used in a system of this type but there are problems finding algorithms that scale with the number of processors which limits their usefulness for general purpose computing.

The most common examples of this architecture are the Meiko Compute Surface, which has Transputer, Intel 860 or SPARC processors; the Intel iPSC Hypercube which has Intel 386, 486 or 860 processors; a myriad of other Transputer based machines and more recently the Thinking Machines CM-5 which has SPARC processors. Most of the above have vector units to provide very high peak floating point performance for running specially written numerical programs very fast. Some have also been used to run databases, in particular the Oracle Parallel Server product. There is no software compatibility across these machines and there is no dominant operating system for this type of computer (although there is a trend towards providing various forms of Unix compatibility). They are often interfaced to a front end Sun workstation that provides a user interface, development environment, disk storage and network interface for the machine.

Shared Memory Multiprocessors

Figure 2 Typical Small Scale Shared Memory Multiprocessor



A shared memory multiprocessor is much more tightly integrated and consists of a fairly conventional starting point of CPU, memory and I/O subsystem with extra CPUs added onto the central bus. This multiplies the load on the memory system by the number of processors and the shared bus becomes a bottleneck. To reduce the load caches are always used and very fast memory systems and buses are built. If more and faster processors are added to the design the cache size needs to increase, the memory system needs to be improved and the bus needs to be speeded up. Most small scale MP machines support up to four processors. Larger ones support a few tens of processors. With some workloads the bus or memory system will saturate before the maximum number of processors has been configured.

Special circuitry is used to snoop activity on the bus at all times so that all the caches can be kept coherent. If the current copy of some data is in more than one of the caches then it will be marked as being shared. If it is updated in one cache then the copies in the other caches are either invalidated or updated automatically. From a software point of view there is never any need to explicitly copy data from one processor to another and shared memory locations are used to communicate values between CPUs. The cache to cache transfers still occur when two CPUs use data in the same cache line so they must be considered from a performance point of view, but the software does not have to worry about it.

There are many examples of shared memory mainframes and minicomputers. Some examples of SPARC based Unix multiprocessors include four processor Sun SPARCserver 600 and SPARCstation 10 models and ICL DRS6000s, eight processor SPARC machines from Solbourne and Cray (the old SPARC based FPS 500EA is sold as the Cray S-MP) and the 20 processor SPARCcenter 2000.

The high end machines often have multiple I/O subsystems and multiple memory subsystems connected to the central bus. This allows more CPUs to be configured without causing bottlenecks in the I/O and memory systems. The SPARCcenter 2000 takes this further by having dual buses with a 256 byte interleave.

Unix On Shared Memory Multiprocessors

Critical Regions

The Unix kernel has many critical regions, or sections of code where a data structure is being created or updated. These regions must not be interrupted by a higher priority interrupt service routine. The uniprocessor Unix kernel manages these regions by setting the interrupt mask to a high value during the region. On a multiprocessor there are other processors with their own interrupt masks so a different technique must be used to manage critical regions.

The Spin Lock Or Mutex

One key capability in shared memory multiprocessor systems is the ability to perform interprocessor synchronisation using atomic load/store or swap instructions. In all SPARC chips there is an instruction called LDSTUB, which means load-store-unsigned-byte. It reads a byte from memory into a register then writes 0xFF into memory in a single indivisible operation. The value in the register can then be examined to see if it was already 0xFF, which means that another processor got there first, or if it was 0x00, which means that this processor is in charge. This is used to make *mutual exclusion locks* (known as mutexes) which make sure that only one processor at a time can hold the lock. The lock is acquired using LDSTUB and cleared by storing 0x00 back to memory. If a processor does not get the lock then it may decide to *spin* by sitting in a loop, testing the lock, until it becomes available. By checking with a normal load instruction in a loop before issuing a LDSTUB the spin is performed within the cache and the bus snooping logic watches for the lock

being cleared. In this way spinning causes no bus traffic so processors that are waiting do not slow down those that are working. A spin lock is appropriate when the wait is expected to be short. If a long wait is expected the process should sleep for a while so that a different job can be scheduled onto the CPU.

Code Locking And SunOS 4.1.X

The simplest way to convert a Unix kernel that is using interrupt levels to control critical regions for use with multiprocessors is to replace the call that sets interrupt levels high with a call to acquire a mutex lock. At the point where the interrupt level was lowered the lock is cleared. In this way the same regions of code are locked for exclusive access. This method has been used to a greater or lesser extent by most MP Unix implementations including SunOS 4.1.2 and SunOS 4.1.3 on the SPARCserver 600MP machines[1], ICL's DRS/NX and Solbourne's OS/MP. The amount of actual concurrency that can take place in the kernel is controlled by the number and position of the locks.

In SunOS 4.1.2 and SunOS 4.1.3 there is effectively a single lock around the entire kernel. The reason for using a single lock is to make these MP systems totally compatible with user programs and device drivers written for uniprocessor systems. User programs can take advantage of the extra CPUs but only one of the CPUs can be executing kernel code at a time.

When code locking is used there are a fixed number of locks in the system and this number can be used to characterise how much concurrency is available. On a very busy, highly configured system the code locks are likely to become bottlenecks so that adding extra processors will not help performance and may actually reduce performance.

Data Locking And Solaris 2.0

The problem with code locks is that different processors often want to use the same code to work on different data. To allow this to happen locks must be placed in data structures rather than code. Unfortunately this requires an extensive rewrite of the kernel which is one reason why Solaris 2 took several years to create[2]. The result is that the kernel has a lot of concurrency available

1. The multiprocessor features are described in "New Technology For Flexibility, Performance And Growth, The SPARCserver 600 Series".

and can be tuned to scale well with large numbers of processors. The same kernel is used on uniprocessors and multiprocessors so all device drivers and user programs must be written to work in the same environment and there is no need to constrain concurrency for compatibility with uniprocessor systems. The locks are still needed in a uniprocessor since the kernel can switch between kernel threads at any time to service an interrupt.

When data locking is used there is a lock for each instance of a data structure. Since table sizes vary dynamically the total number of locks grows as the tables grow and the amount of concurrency that is available to exploit is greater on a very busy, highly configured system. Adding extra processors to such a system is likely to be beneficial. Solaris 2 has about 150 different data locks and multiplied by the number of instances of the data there will typically be several thousand locks in existence on a running system.

SPARC Based Multiprocessor Hardware

Bus Architectures

There are two things to consider about bus performance. The peak data rate is easily quoted but the ability of the devices on the bus to source or sink data at that rate for more than a few cycles is the real limit to performance. The second thing to consider is whether the bus protocol includes cycles that do not transfer data which reduces the sustained data throughput.

Older buses like VMEbus usually transfer one word at a time so that each bus cycle includes the overhead of deciding which device will access the bus next (arbitration) as well as setting up the address and transferring the data. This is rather inefficient so more recent buses like SBus and MBus transfer data in blocks. Arbitration can take place once per block then a single address is set up and multiple cycles of data are transferred. The protocol gives better throughput if more data is transferred in each bus transaction. For example SPARCserver 600MP systems are optimised for a standard transaction size of

2. See “Solaris 2.0 Multithread Architecture White Paper” and “Realtime Scheduling In SunOS 5.0, Sandeep Khanna, Michael Sebrée, John Zolnowsky”.

32 bytes by providing 32 byte buffers in all the devices that access the bus and using a 32 byte cache line[1]. The SPARCcenter 2000 is optimised for 64 byte transactions[2].

Circuit Switched Bus Protocols

One class of bus protocols effectively opens a circuit between the source and destination of the transaction and holds on to the bus until the transaction has finished and the circuit is closed. This is simple to implement but when a transfer from a slow device like main memory to a fast device like a CPU cache (a cache read) occurs there must be a number of wait states to let the main memory DRAM access complete in between sending the address to memory and the data returning. These wait states reduce cache read throughput and nothing else can happen while the circuit is open. The faster the CPU clock rate the more clock cycles are wasted. On a uniprocessor this adds to the cache miss time which is annoying but on a multiprocessor the number of CPUs that a bus can handle is drastically reduced by the wait time. Note that a fast device like a cache can write data with no delays to the memory system write buffer. MBus uses this type of protocol and is suitable for up to four CPUs.

Packet Switched Bus Protocols

To make use of the wait time a bus transaction must be split into a request packet and a response packet. This is much harder to implement because the response must contain some identification and a device on the bus such as the memory system may have to queue up additional requests coming in while it is trying to respond to the first one. There is a protocol extension to the basic MBus interface called XBus that implements a packet switched protocol on the SuperCache controller chip used with SuperSPARC. This provides more than twice the throughput of MBus and it is designed to be used in larger multiprocessor machines that have more than four CPUs on the bus. The SPARCcenter 2000 uses XBus within each CPU board and multiple interleaved

1. See the “SPARCserver 10 and SPARCserver 600 White Paper” for details of the buffers used.

2. See “The SPARCcenter 2000 Architecture and Implementation White Paper”.

XBuses on its interboard backplane. The backplane bus is called XDbus and on the SPARCcenter 2000 there is a twin XDbus (one, two or four could be implemented with minor changes to the chipset and different board sizes).

Table 19 MP Bus Characteristics

Bus Name	Peak Bandwidth	Read Throughput	Write Throughput
Solbourne KBus	128Mbytes/s	66Mbytes/s	90Mbytes/s
ICL HSPBus	128Mbytes/s	66Mbytes/s	90Mbytes/s
MBus	320Mbytes/s	90Mbytes/s	200Mbytes/s
XBus	320Mbytes/s	250Mbytes/s	250Mbytes/s
Dual XDbus	640Mbytes/s	500Mbytes/s	500Mbytes/s

MP Cache Issues

In systems that have more than one cache on the bus a problem arises when the same data is stored in more than one cache and the data is modified. A cache coherency protocol and special cache tag information is needed to keep track of the data. The basic solution is for all the caches to include logic that watches the transactions on the bus (known as snooping the bus) and look for transactions that use data that is being held by that cache[1]. The I/O subsystem on Sun's multiprocessor machines has its own MMU and cache so that full bus snooping support is provided for DVMA[2] I/O transfers. The coherency protocol that MBus defines uses invalidate transactions that are sent from the cache that owns the data when the data is modified. This invalidates any other copies of that data in the rest of the system. When a cache tries to read some data the data is provided by the owner of that data, which may not be the memory system, so cache to cache transfers occur. An MBus option which is implemented in Sun's MBus based system is that the memory system grabs a copy of the data as it goes from cache to cache and updates itself[3].

1. This is described very well in "SPARCserver 10 and SPARCserver 600 White Paper".

2. DVMA stands for Direct Virtual Memory Access and it is used by intelligent I/O devices that write data directly into memory using virtual addresses e.g. the disk and network interfaces.

3. This is known as *reflective memory.*

The cache coherency protocol slows down the bus throughput slightly compared to a uniprocessor system with a simple uncached I/O architecture which is reflected in extra cycles for cache miss processing. This is a particular problem with the Ross CPU module used in the first generation SPARCserver 600 systems. The cache is organised as a 32 bit wide bank of memory while the MBus transfers 64bit wide data at the full cache clock rate. A 32 byte buffer in the 7C605 cache controller takes the data from the MBus then passes it onto the cache. This extra level of buffering increases cache miss cost but makes sure that the Mbus is freed early to start the next transaction. This also makes cache to cache transfers take longer. The SuperSPARC with SuperCache module has a 64bit wide cache organisation so the intermediate buffer is not needed and cache miss costs and cache to cache transfer times are much reduced. This extra efficiency helps the existing bus and memory system cope with the extra load generated by CPUs that are two to three times faster than the original Ross CPUs and future SuperSPARC modules that may double performance again.

Memory System Interleave On The SPARCcenter 2000

The SC2000 has up to ten system boards with two memory banks on each board. Each bank is connected to one of the XDBuses. At boot time the system configures its physical memory mappings so that the memory systems on each XDBus are interleaved together on 64 byte boundaries as well as the 256 byte interleave of the dual XDBus itself. The maximum combined interleave occurs when eight memory banks are configured, with four on each XDBus. This is a minimum of 512 Mb of RAM and higher performance can be obtained using eight 64Mb banks rather than two 256Mb banks. The effect of the interleave can be considered for the common operation of a 4 Kb page being zeroed. The first four 64 byte cache lines are passed over the first XDBus to four different memory banks, the next four 64 byte cache lines are passed over the second XDBus to four more independent memory banks. After the first 512 bytes has been written the next write goes to the first memory bank again, which has had plenty of time to store the data into one of the special SIMMs. On average the access pattern of all the CPUs and I/O subsystems will be distributed randomly across all of the (up to twenty) memory banks, the interleave prevents a block sequential access from one device from hogging all the bandwidth in any one memory bank.

In summary, use more system boards with 64Mb memory options on the SC2000 to give better performance than fewer 256Mb memory options and install the memory banks evenly across the two XDBuses as far as possible.

Measuring And Tuning A Multiprocessor

SunOS 4.1.2 and 4.1.3

There are two “MP aware” commands provided. /usr/kvm/mpstat breaks down the CPU loading for each processor and a total. /usr/kvm/mps is the same as ps (1), in that it lists the processes running on the machine except that it includes a column to say which CPU each process was last scheduled onto.

Understanding Vmstat On A Multiprocessor

Vmstat provides the best summary of performance on an MP system. If there are two processors then a CPU load of 50% means that one processor is completely busy. If there are four processors then a CPU load of 25% means that one processor is completely busy.

The first column of output is the average number of runnable processes. To actually use two or four processors effectively this number must average more than two or four. On a multiuser timesharing or database server this figure can be quite large. If you currently have a uniprocessor or dual processor system and are trying to decide whether an upgrade to two or four processors would be effective this is the first thing you should measure. If you have a compute server that only runs one job then it may use 100% of the CPU on a uniprocessor but it will only show one runnable process. On a dual processor this workload would only show 50% CPU busy. If there are several jobs running then it may still show 100% CPU busy but the average number of runnable processes will be shown and you can decide how many CPUs you should get for that workload.

The number of context switches is reduced on a multiprocessor as you increase the number of CPUs. For some workloads this can increase the overall efficiency of the machine. One case when the context switch rate will not be reduced can be illustrated by considering a single Oracle database batch job. In this case an application process is issuing a continuous stream of database queries to its Oracle back-end process via a socket connection between the two. There are two processes running flat out and context switch rates can be very high on a uniprocessor. If a dual processor is tried it will not help. The two processes never try to run at the same time, one is always waiting for the other one, so on every front-end to back-end query there will be a context switch

regardless. The only benefit of two processors is that each process may end up being cached on a separate processor for an improved cache hit rate. This is very hard to measure directly.

The number of interrupts, system calls and the system CPU time are the most important measures from a tuning point of view. When a heavy load is placed on a SPARCserver600MP the kernel may bottleneck on system CPU time of 50% on a two processor or 25% on a four processor. The number of interrupts and system calls and the time taken to process those calls must be reduced to improve the throughput. Some kernel tweaks to try are increasing maxslp to prevent swap outs and reducing slowscan and fastscan rates. If you can reduce the usage of system call intensive programs like find or move a database file from a filesystem to a raw partition it will also help. There is a patch for 4.1.2 (100575) which reduces spinning by making disk I/O more efficient (reducing the system time) and allows some kernel block copies to occur outside the mutex lock so that more than one CPU can be productive (fixed in 4.1.3).

It is hard to see how much time each CPU spends spinning waiting for the kernel lock but the total productive system CPU time will never exceed one processors worth so any excess CPU time will be due to spinning. E.g. on a four processor machine reporting a sustained 75% system CPU time 25% is productive (one CPUs worth) and 50% is spent waiting for the kernel lock (two CPUs worth that might as well not be configured). Conversely a two processor machine that often has system CPU time over 50% will not benefit from adding two more processors and may actually slow down. It is possible to patch a kernel so that it will ignore one or more CPUs at boot time. This allows testing on variable numbers of processors without having to physically remove the module. The okprocset kernel variable needs to be patched in a copy of vmunix using adb then the system must be rebooted using the patched kernel. These values assume that CPU #1 is in control and okprocset controls which additional CPUs will be started. Check how many you have with `mpstat`.

Table 20 Disabling Processors In SunOS 4.1.X

Number Of CPU's	adb -w /vmunix.Ncpu
1	okprocset?W0x1
2	okprocset?W0x3
3	okprocset?W0x7
4	okprocset?W0xf

Solaris 2

Solaris 2.0 only runs on uniprocessor machines (Sun4c kernels). Solaris 2.1 has been tested and tuned for machines with up to four CPUs. Preliminary tests show that high system time percentages can be sustained productively (without spinning). Performance figures for Solaris 2 are contained in the "SPARCserver and SPARCcenter Performance Brief". The Solaris 2.2 release during the first half of 1993 will support the SPARCcenter 2000 in configurations of up to eight CPUs. The subsequent release during the second half of 1993 will support the full twenty CPU configuration.

Cache Affinity Algorithms

When a system that has multiple caches is in use a process may run on a CPU and load itself into that cache than stop running for a while. When it resumes the Unix scheduler must decide which CPU to run it on. To reduce cache traffic the process must preferentially be allocated to its original CPU, but that CPU may be in use or the cache may have been cleaned out by another process in the meantime. The cost of migrating a process to a new CPU depends upon the time since it last ran, the size of the cache and the speed of the central bus. A very delicate balance must be struck and a general purpose algorithm that adapts to all kinds of workloads must be developed. In the case of the SPARCcenter 2000 this algorithm is managing up to 20Mbytes of cache and will have a very significant effect on multiuser performance. The development, tuning and testing of the algorithm used in Solaris 2 will take place during 1993 and is one reason for the staged introduction of the SPARCcenter 2000.

Programming A Multiprocessor

SunOS 4.1.2 and 4.1.3

There is no supported programmer interface to the multiple processors. Multiple Unix processes must be used.

Solaris 2.0 and 2.1

A multi-threaded programmer interface has been designed[1] but it has not yet been released. The main delay is that both Unix International (SVR4 ES/MP) and a POSIX standards committee (1003.4a) are deciding on standards for

multi-threading and Sun wishes to conform to the standards rather than release a library that would become obsolete as soon as the standards are finished.

Deciding How Many CPU's To Configure

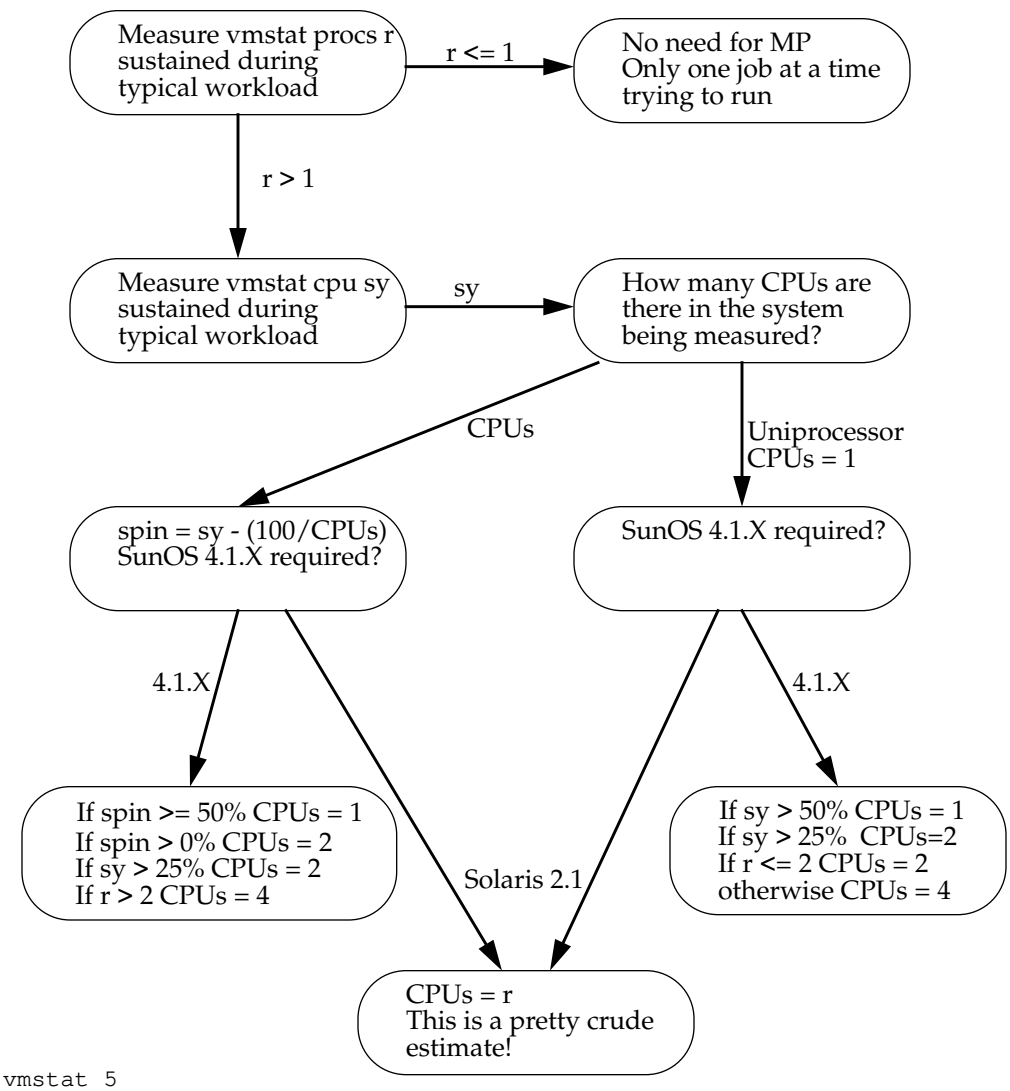
If you have an existing machine running a representative workload then you can look at the vmstat output to come up with an approximate guess for the number of CPU's that could be usefully employed by the workload. Interpreting vmstat depends upon the number of CPU's configured in the existing machine and whether it is running SunOS 4.X or Solaris 2.X. The first line of vmstat output should be ignored and some feel for the typical values of CPU system time percentage and the number of processes in the run queue should be used as the basis for estimation. A flow chart is provided overleaf to guide you through this process. The results should be used as input into a decision and should not be taken too seriously. This methodology has been tried out a few times with some success and any feedback confirming or contradicting the method would be welcome.

Vmstat Run Queue Differences In Solaris 2

It is important to note that vmstat in SunOS 4.X reports the number of jobs in the run queue including the jobs that are actually running. Solaris 2 only reports the jobs that are waiting to run. The difference is equal to the number of CPUs in the system. A busy 4 CPU system might show six jobs in the run queue on SunOS 4.1.3 and only two jobs on Solaris 2 even though the two systems are behaving identically. If vmstat reports zero jobs in Solaris 2 then there may be one, two, three or four jobs running and you have to work it out for yourself from the CPU percentages. If you use a Solaris 2 measured workload with the flowchart overleaf then you need to add the number of CPUs in the system to `r` if `r` is non-zero, or work out `r` from the CPU times if it is zero.

1. See "Solaris 2.0 Multithread Architecture White Paper".

Maximum Number Of CPU's For A Measured SunOS 4.X Workload




```
% vmstat 5
 procs      memory                 page                 disk        faults      cpu
 r b w    avm    fre   re at   pi   po   fr   de   sr s0 s0 s1 s2   in   sy   cs us sy id
 0 1 0      0 21992    0  3    3    0    0    0    0  0  1  0  0   25 100   15  7  3 91
 2 0 0      0 21928    0  0    0    0    0    0    0  0  7  0  0  176 222   40 29  9 62
```

Network

This topic has been extensively covered in other white papers. An overview of the references is provided at the end of this chapter.

The network throughput tuning performed for SunOS 4.1

There were two major changes in SunOS 4.1, the internal buffer allocation code was changed to handle FDDI packets (4.5Kb) more effectively and the TCP/IP and ethernet device driver code was tuned. The tuning followed suggestions made by Van Jacobson[1].

When routing packets from one ethernet to another a SPARCstation 1 running SunOS 4.0.3 could route up to 1815 packets/second from one ethernet interface to another. After tuning this increased to 6000 packets/second using SunOS 4.1. The SPARCstation 2 running SunOS 4.1.1 can route 12000 packets/second. These are all 64 byte minimum sized packets, where the overhead is greatest.

Different ethernet interfaces and how they compare

There are two main types of ethernet interface used on Sun machines, the Intel (ie) and AMD LANCE (le) chips. This is further complicated by the way that the ethernet chip is interfaced to the CPU, it may be built into the CPU board or interfaced via SBus, VMEbus or Multibus. The routing performance of various combinations has been measured and is shown in table 10-1.

SBus Interface - le

This interface is used on all SPARC desktop machines. The built-in ethernet interface shares its DMA connection to the SBus with the SCSI interface but has higher priority so heavy ethernet activity can reduce disk throughput. This can

1. "An Analysis of TCP Processing Overhead, by David Clark, Van Jacobson, John Romkey, Howard Salwen, June 1989, IEEE Communications

be a problem with the original DMA controller used in the SPARCstation 1, 1+, SLC and IPC but subsequent machines have enough DMA bandwidth to support both. The add-on SBus ethernet card uses exactly the same interface as the built-in ethernet, but it has an SBus DMA controller to itself. The more recent buffered ethernet interfaces used in the SPARCserver 600, the SBE/S, the FSBE/S and the DSBE/S, have a 256Kbyte buffer to provide a low latency source and sink for the ethernet. This cuts down on dropped packets, especially when many ethernets are configured in a system that also has multiple CPU's consuming the memory bandwidth.

Built-in Interface - le0

This interface is built into the CPU board on the SPARCsystem 300 range and provides similar throughput to the SBus interface.

Built-in Interface - ie0

This interface is built into the CPU board on the Sun4/110, Sun4/200 range and SPARCsystem 400 range. It provides reasonable throughput but is generally slower than the "le" interface.

VMEbus Interface - ie

This is the usual board provided on SPARCsystem 400 series machines to provide a second, third or fourth ethernet as ie2, ie3 and ie4. It has some local buffer space and is accessed over the VMEbus. It is not as fast as the on-board "ie" interface but it does use 32 bit VMEbus transfers. The SPARCserver 400 series has a much faster VMEbus interface than other Sun's which helps this board perform better.

Multibus Interface - ie

This is an older board which uses a Multibus to VME adapter so all VME transfers happen at half speed as 16 bit transfers and it is rather slow. It tends to be configured as ie1 but it should be avoided if a lot of traffic is needed on that interface.

VMEbus Interphase NC400 - ne

This is an intelligent board which performs ethernet packet processing up to the NFS level. It can support about twice as many NFS operations per second as the built-in “ie0” interface on a SPARCserver 490. It actually supports fewer NFS operations per second than “le” interfaces but off-loads the main CPU so that more networks can be configured. There are no performance figures for routing throughput. It only accelerates the UDP protocol used by NFS and TCP is still handled by the main CPU.

Routing Throughput

The table below shows throughput between various interfaces on SPARCstation 1 and Sun4 /260 machines. The Sun4/260 figures are taken from an internal Sun document by Neal Nuckolls. The SPARCstation figures are in the SPARCstation 2 Performance Brief.

Table 21 Ethernet Routing Performance

Machine	From	To	64 byte packets/sec
Sun4/260	On-board ie0	Multibus ie1	1813
Sun4/260	Multibus ie1	On-board ie0	2219
Sun4/260	On-board ie0	VMEbus ie2	2150
Sun4/260	VMEbus ie2	On-board ie0	2435
SPARCstation1	On-board le0	SBus le1	6000
SPARCstation 2	On-board le0	SBus le1	12000

Using NFS effectively

- “Managing NFS and NIS, Hal Stern, O’Reilly”, essential reading!
- “Networks and File Servers: A Performance Tuning Guide”
- “SPARCserver 490 NFS File Server Performance Brief, February 1991”
- “Tuning the SPARCserver 490 for Optimal NFS Performance, February 1991”
- “SunOS 4.1 Performance Tuning, by Hal Stern”.

References A

2.1Gb 5.25-inch Fast Differential SCSI-2 Disk Products

Sun's 2.1Gb disk drive introduces several new technologies that are described by this paper. Fast SCSI at 10Mb/s, differential drive to provide 25 meter cable lengths, and tagged command queueing to optimise multiple commands inside the drive are discussed in detail.

Administering Security, Performance and Accounting in Solaris 2.0

This is the basic reference on Solaris 2.0 which is part of the manual set or AnswerBook CD. It describes the tweakable parameters that can be set in /etc/system and provides a tutorial on how to use the performance analysis tools such as sar and vmstat.

An Analysis of TCP Processing Overhead, by David Clark, Van Jacobson, John Romkey, Howard Salwen, June 1989, IEEE Communications

This paper describes the things that can be tuned to improve TCP/IP and ethernet throughput.

Building and Debugging SunOS Kernels, by Hal Stern

This Sun white paper provides an in-depth explanation of how kernel configuration works, with some performance tips.

A

A Cached System Architecture Dedicated for the System IO Activity on a CPU Board, by Hseih, Wei and Loo.

This is published in the proceedings of the 1989 International Conference on Computer Design, October 2 1989. It describes the patented Sun I/O cache architecture.

Computer Architecture - A Quantitative Approach, Hennessy and Patterson

The definitive reference book on modern computer architecture.

Extent-like Performance from a Unix File System, L. McVoy and S. Kleiman

This paper was presented at Usenix in Winter 1991. It describes the file system clustering optimisation that was introduced in SunOS 4.1.1.

Graphics Performance Technical White Paper, January 1990

This provides extensive performance figures for PHIGS and GKS running a wide variety of benchmarks on various hardware platforms. It is becoming a little out of date as new versions of PHIGS, GKS and hardware have superseded those tested but is still very useful.

Managing NFS and NIS, Hal Stern, O'Reilly

Network administration and tuning is covered in depth. An essential reference.

Networks and File Servers: A Performance Tuning Guide

This is the best of the network tuning white papers.

New Technology For Flexibility, Performance And Growth, The SPARCserver 600 Series

This white paper, despite its overblown title, provides a very good technical overview of the architecture of the hardware and how SunOS 4.1.2 works on a multiprocessor system. It also describes the SPARC reference MMU in some detail in an appendix. A new version called “SPARCserver 10 and SPARCserver 600 White Paper” is also available.

OpenWindows Version 2 White Papers

This includes `x11perf` results for OpenWindows 2 compared with MIT X11R4 and DECwindows. This is now rather out of date.

Optimisation in the SPARCompilers, Steven Muchnick

The definitive description of the SPARC compilers and how they work.

Realtime Scheduling In SunOS 5.0, Sandeep Khanna, Michael Sebrée, John Zolnowsky

This paper was presented at Usenix Winter ‘92 and describes the implementation of the kernel in Solaris 2.0. It covers how kernel threads work, how real-time scheduling is performed and the novel priority inheritance system.

SBus Specification Rev B

This is the latest version of the SBus specification, including 64bit extensions. It is available free from Sun sales offices and from the IEEE as proposed standard IEEE P1496.

SCSI-2 Terms, Concepts, and Implications Technical Brief

There is much confusion and misuse of terms in the area of SCSI buses and standards. This paper clears up the confusion.

SCSI And IPI Disk Interface Directions

After many years where IPI has been the highest performance disk technology the time has come where SCSI disks are not only less expensive but are also higher performance and SCSI disks with on-disk buffers and controllers are beginning to include the optimisations previously only found in IPI controllers.

The SPARC Architecture Manual Version 8

This is available as a book from Prentice Hall. It is also available from the IEEE P1754 working group. Version 9 of the SPARC Architecture is in the process of being published and includes upwards compatible 64bit extensions and a revised kernel level interface.

Solaris 2.0 Multithread Architecture White Paper

The overall multiprocessor architecture of Solaris 2.0 is described. In particular the user level thread model is explained in detail. This paper has also appeared as “SunOS Multithread Architecture” at Usenix Winter ‘91.

The SPARCcenter 2000 Architecture and Implementation White Paper

This paper contains an immense amount of detail on this elegant but sophisticated large scale multiprocessor server. A good understanding of computer architecture is assumed.

SPARC Compiler Optimisation Technology Technical White Paper

This describes the optimisations performed by Sun’s compilers. If you know what the compiler is looking for it can help to guide your coding style.

SPARCengine IPX User Guide

This can be ordered from Sun using the product code SEIPX-HW-DOC. It provides the full programmers model of the SPARCstation IPX CPU board including internal registers and address spaces and a full hardware description of the board functions.

SPARCengine 2 User Guide

This can be ordered from Sun using the product code SE2-HW-DOC. It provides the full programmers model of the SPARCstation 2 CPU board including internal registers and address spaces and a full hardware description of the board functions.

SPARCengine 10 User Guide

This can be ordered from Sun using the product code SE10-HW-DOC. It provides the full programmers model of the SPARCstation 10 CPU board including internal registers and address spaces and a full hardware description of the board functions.

SPARC Strategy and Technology, March 1991

This contains an excellent introduction to the SPARC architecture, as well as the features of SunOS 4.1.1, the migration to SVR4 and the SPARC cloning strategy. It is available from Sun sales offices that are not out of stock.

SPARC Technology Conference notes - 3 Intense Days of Sun

The SPARC Technology Conference toured the world during 1991 and 1992. It was particularly aimed at SPARC hardware and real time system developers. You needed to go along to get the notes!

SPARCserver 490 NFS File Server Performance Brief, February 1991

This white paper compares performance of the SS490 with Prestoserve and Interphase NC400 ethernet interfaces against the Auspex NS5000 dedicated NFS server.

SPARCserver 10 and SPARCserver 600 White Paper

This is an update of “New Technology For Flexibility, Performance And Growth, The SPARCserver 600 Series” to include the SPARCserver 10.

SPARCserver Performance Brief

This provides benchmark results for multi-user, database and file-server operations on SPARCserver 2 and SPARCserver 400 series machines running SunOS 4.1.1.

SPARCserver 10 and SPARCserver 600 Performance Brief

The standard benchmark performance numbers for SuperSPARC based SPARCserver 10 and SPARCserver 600 systems running SunOS 4.1.3 are published in this document.

SPARCserver and SPARCcenter Performance Brief

The performance of the SPARCcenter 2000 is measured with varying numbers of processors and is compared to the SPARCserver 10 and SPARCclassic server running parallelised SPEC92, SPECthroughput, AIM III and parallelised Linpack benchmarks using the Solaris 2.1 operating system.

SPARCstation 2GS / SPARCstation 2GT Technical White Paper

This describes the 3D graphics accelerators available from Sun.

SPARCstation 2 Performance Brief

This provides benchmark results for the SPARCstation 2 running SunOS 4.1.1, comparing it against other Sun's and some competitors.

SPARCstation 10 White Paper

A comprehensive overview of the SPARCstation 10.

Sun Systems and their Caches, by Sales Tactical Engineering June 1990.

This explains Sun caches in great detail, including the I/O cache used in high end servers. It does not include miss cost timings however.

SunOS 4.1 Performance Tuning, by Hal Stern

This white paper introduces many of the new features that were introduced in SunOS 4.1. It complements this overview document as I have tried to avoid duplicating detailed information so it is essential reading although it is getting out of date now.

SunOS System Internals Course Notes

These notes cover the main kernel algorithms using pseudo-code and flowcharts, to avoid source code licencing issues. The notes are from a 5 day course which is run occasionally at Sun UK.

SunPHIGS / SunGKS Technical White Paper

There is little explicit performance information in this paper but it is a very useful overview of the architecture of these two graphics standards and should help developers choose an appropriate library to fit their problems.

The SuperSPARC Microprocessor Technical White Paper

This paper provides an overview to the architecture, internal operation and capabilities of the SuperSPARC microprocessor used in the SPARCstation 10, SPARCserver 600 and SPARCcenter 2000 machines.

System Performance Tuning, Mike Loukides, O'Reilly

This is an excellent reference book that covers tuning issues for SunOS 4, BSD, System V.3 and System V.4 versions of Unix. It concentrates on tuning the operating system as a whole, particularly for multi-user loads and contains a lot of information on how to manage a system, how to tell which part of the system may be overloaded, and what to tweak to fix it. ISBN 0-937175-60-9.

tmpfs: A Virtual Memory File System, by Peter Snyder

The design goals and implementation details are covered by this Sun white paper.

Tuning the SPARCserver 490 for Optimal NFS Performance, February 1991

This paper describes the results of some NFS performance testing and provides details of scripts to modify and monitor the buffer cache.

XGL Graphics Library Technical White Paper

A standard Sun white paper describing the highest performance graphics library available on Sun systems.

You and Your Compiler, by Keith Bierman

This is the definitive guide to using the compilers effectively for benchmarking or performance tuning. It has been incorporated into the manual set for Fortran 1.4 in a modified form as a performance tuning chapter.

Revision History

Revision	Dash	Date	Comments
8xx-xxxx-xx	-01	July 1991	First complete draft for review
8xx-xxxx-xx	-02	August 1991	Cache details corrected
8xx-xxxx-xx	-03	September 1991	More corrections, presented at Sun User '91, distributed
8xx-xxxx-xx	-04	September 1992	Update to 4.1.3, S600MP and SS10, draft issued internally
8xx-xxxx-xx	-05	December 1992	Update to Solaris 2.1, SC2000, LX and Classic, distributed

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
415 960-1300
FAX 415 969-9131

For U.S. Sales Office locations, call:
800 821-4643
In California:
800 821-4642

Australia: (02) 413 2666
Belgium: 32-2-759 5925
Canada: 416 477-6745
Finland: 358-0-502 27 00
France: (1) 30 67 50 00
Germany: (0) 89-46 00 8-0
Hong Kong: 852 802 4188
Italy: 039 60551
Japan: (03) 221-7021
Korea: 822-563-8700
Latin America: 415 688-9464
The Netherlands: 033 501234
New Zealand: (04) 499 2344
Nordic Countries: +46 (0) 8 623 90 00
PRC: 861-831-5568
Singapore: 224 3388
Spain: (91) 5551648
Switzerland: (1) 825 71 11
Taiwan: 2-514-0567
UK: 0276 20444

Elsewhere in the world,
call Corporate Headquarters:
415 960-1300
Intercontinental Sales: 415 688-9000