Sun Performance Tuning Overview

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

SMCC Technical Marketing

December 1993

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

Part No.: 801-4872-07
Revision B, December 1993

Contents

Preface xiii

1. Introduction 15

Performance Measurement 16

Quick Reference for Common Tuning Tips 17

The First 10 Tuning Steps 17

2. Source Code 19

Algorithms 19

Algorithmic Classification 19

An Example Problem 20

Space Versus Time 20

Programming Model 21

Choice Of Language To Express The Algorithm Or Model 21

Debug And Test Tools 23

Compiler And Optimisations 24

Automatic Parallelization 24

Effective Use Of Provided System Functions 25
Mapped files 25
Asynchronous I/O 25
Memory Allocation Tuning 26
Linking, Localisation Of Reference And Libraries 26
Tuning Shared Libraries 27
3. Executable 29
Customising The Execution Environment 29
Limits 29
Tracing In SunOS 4.X 30
Tracing In Solaris 2 32
Timing 33
The Effects Of Underlying Filesystem Type 33
UFS 34
Tmpfs 34
NFS 35
Cachefs 35
Tuning Filesystems 36
Tunefs 36
Eagle DiskPak 36
4. Databases & Configurable Systems 37
Examples 37
Hire An Expert! 37
Use Sun's Database Excelerator Product with SunOS 4.X 37

Basic Tuning Ideas 38
Increasing Buffer Sizes 38
Using Raw Disk Rather Than Filesystems 38
Balance The Load Over All The Disks 39
Which Disk Partition To Use 39
The Effect Of Indices 40
5. Kernel 41
Buffer Sizes And Tuning Variables 41
Solaris 2 Performance Tuning Manual 42
Maxusers In SunOS 4.X. 42
Using /etc/system To Modify Kernel Variables In Solaris 2 42
Maxusers In Solaris 2 43
Using Pstat In SunOS 4.X To Examine Table Sizes 44
Using Sar In Solaris 2 To Examine Table Sizes And Kernel Memory Allocation 45
Directory Name Lookup Cache (dnlc) 45
Inode Cache 46
Buffer Cache 46
Setting Default Limits 48
Solaris 2 Performance Improvements 49
Using Sar Effectively In Solaris 2. 50
MMU Algorithms And PMEGS 50
The Sun-4 MMU - sun4, sun4c, sun4e Kernel Architectures 50
Contexts 52

The SPARC Reference MMU - sun4m Kernel Architecture 53
The SPARC Reference MMU Table Walk Operation 54
The Paging Algorithm and How To Tune It 56
Understanding Vmstat Output 56
The Paging Algorithm In SunOS 4.X 58
Kernel Variables To Control Paging In SunOS 4.X 59
Swapping Out And How To Get Back In (SunOS 4.X) 62
The Paging Algorithm In Solaris 2 63
Swapping Out And How To Get Back In (Solaris 2) 67
Sensible Tweaking 67
Configuring Devices 72
Kernel Configuration In SunOS 4.X. 72
Kernel Configuration In Solaris 2 72
Mapping Device Nicknames To Full Names In Solaris 2 . . 72
Kernel Profiling Using Kgmon In Solaris 2. 74
Monitoring The System With Proctool 76
References 77
6. Memory. 79
Cache Tutorial 79
Why Have A Cache? 79
Cache Line And Size Effects 80
A Problem With Linked Lists 82
Cache Miss Cost And Hit Rates For Different Machines 84
Virtual Write Through Caches 84

Virtual Write Back Caches 85
Physical Write Back Caches 86
On-Chip Caches 87
The SuperSPARC Two Level Cache Architecture 88
I/O Caches 90
Kernel Block Copy 90
Software Page Copies 91
The Classic Cache Aligned Block Copy Problem. 91
Kernel Bcopy Acceleration Hardware. 92
7. Windows and Graphics 93
8. Disk. 95
The Disk Tuning Performed For SunOS 4.1.1. 95
Throughput Of Various Common Disks 95
Understanding The Specification. 95
Sequential Versus Random Access 99
Effect Of Disk Controller, SCSI, SMD, IPI. 99
SCSI Controllers. 99
SMD Controllers 101
IPI Controllers 101
Load Monitoring And Balancing 102
Iostat In SunOS 4.X 102
Iostat In Solaris 2 103
How To Decide That A Disk Is Overloaded 104
Multiple Disks On One Controller. 106

Mirrored Disks 106
Tuning Options For Mirrors 107
9. CPU 109
Architecture And Implementation. 109
Instruction Set Architecture (ISA) 109
SPARC Implementation 110
System Architecture 110
Kernel Architecture 110
The Effect Of Register Windows And Different SPARC CPUs. 110
The Effect Of Context Switches And Interrupts. 112
Comparing Instruction Cycle Times On Different SPARC CPUs 113
The Weitek SPARC PowerUP Upgrade. 115
Superscalar Operations 115
Low Cost Implementations 116
Floating Point Performance Comparisons 116
Integer Performance Comparisons 118
10. Multiprocessors. 119
Basic Multiprocessor Theory 119
Why Bother With More Than One CPU? 119
Multiprocessor Classifications 119
Unix On Shared Memory Multiprocessors 122
Critical Regions 122
The Spin Lock Or Mutex 122
Code Locking And SunOS 4.1.X 123

Data Locking And Solaris 2.0 123
SPARC Based Multiprocessor Hardware 124
Bus Architectures 124
MP Cache Issues 126
Memory System Interleave On The SPARCcenter 2000 . . . 127
Measuring And Tuning A Multiprocessor With SunOS 4.1.2 and 4.1.3 128
Understanding Vmstat On A Multiprocessor 128
Measuring And Tuning A Multiprocessor With Solaris 2. 130
CPU Control Commands - psrinfo and psradm 130
Cache Affinity Algorithms 130
Programming A Multiprocessor 131
Mp Programming With SunOS 4.1.2 And 4.1.3 131
MP Programming With Solaris 2.0 And 2.1 131
MP Programming With Solaris 2.2 131
MP programming With Solaris 2.3 131
Deciding How Many CPU's To Configure 132
Vmstat Run Queue Differences In Solaris 2 132
Interrupt Distribution Tuning 134
11. Network 135
The Network Throughput Tuning Performed For SunOS 4.1 . 135
Different Ethernet Interfaces And How They Compare 135
SBus Interface - le 135
Built-in Interface - le0 136

Built-in Interface - ie0 136
VMEbus Interface - ie 136
Multibus Interface - ie 136
VMEbus Interphase NC400 - ne. 137
Routing Throughput 137
FDDI Interfaces 137
VMEbus FDDI/DX 138
SBus FDDI/S 138
Using NFS Effectively 139
How Many nfsds? 139
A. References 141

Tables

Table 1	System Broken Down Into Tuning Layers	15
Table 2	Resource Limits	29
Table 3	`Trace` Output with Comments	31
Table 4	Tuning SunOS Releases	41
Table 5	Maxusers Settings In Solaris 2.2	43
Table 6	Default Settings for Kernel Parameters	44
Table 7	Sun-4 MMU Characteristics	51
Table 8	SPARC Reference MMU Characteristics	53
Table 9	Vmstat fields explained	57
Table 10	Application speed changes as hit rate varies with a 25 cycle miss cost	81
Table 11	Virtual Write Through Cache Details	85
Table 12	Virtual Write Back Cache Details	85
Table 13	Physical Write Back Cache Details	86
Table 14	On-Chip Cache Details	88
Table 15	Disk Specifications	97
Table 16	1.3GB IPI ZBR Disk Zone Map	98

Table 17	Which SPARC IU and FPU does your system have?	113
Table 18	Floating point Cycles per Instruction	117
Table 19	Number of Register Windows and Integer Cycles per Instruction 118	
Table 20	MP Bus Characteristics	126
Table 21	Disabling Processors In SunOS 4.1.X	130
Table 22	Ethernet Routing Performance	137

Preface

This paper was originally written for a Sun UK User Group conference in September 1991, it was extensively updated for a second Sun UK User Group conference in January 1993. Both issues of the paper were widely circulated. The first two issues were not officially sanctioned documents and did not have a Sun documentation part number. The author now works for Sun Microsystems Computer Corporation's Technical Marketing group; this document has had many updates and corrections, and it is now an official SMCC white paper with a Sun part number.

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.

This document is intended to be read sequentially as it follows a set structure. Each chapter is largely self contained however, so it can be used as a reference manual if required. The contents pages have been made more detailed in this release, to compensate for the lack of an index!

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. The author is working with SunSoft press and Prentice Hall to produce a book based upon this white paper for publication in mid 1994.

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

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

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

Adrian Cockcroft
SMCC Technical Marketing
Mailstop UPAL1-431
2550 Garcia Avenue
Mountainview
CA 94043
USA

Email: adrian.cockcroft@corp.sun.com

1. "System Performance Tuning, Mike Loukides, O'Reilly"
2. "Managing NFS and NIS, Hal Stern"

Introduction

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 changes 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
Source code	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

Performance Measurement

In many cases it is obvious that there is a performance problem, and when it is fixed there may be a noticeable improvement. The performance tuner may then wrap up a job well done.

Problems can arise when it is necessary to quantify the changes, or to measure minor performance changes in complex systems. If the aim of the work is to improve the performance in a well defined benchmark such as one of the SPEC or TPC measures then the measurement can be well defined. In real life it may be necessary to design a controlled set of experiments to measure performance on the system being worked upon. To distinguish between two sets of results there are statistical tests that can tell you whether the results are significantly different at a particular confidence level.

Experimental design and analysis is beyond the scope of this paper, a recent book on the subject is highly recommended instead. “The Art of Computer Systems Performance Analysis, by Raj Jain” provides comprehensive coverage of techniques for experimental design, measurement, simulation and performance modelling. If you are embarking on a large project that involves a lot of measurement and analysis you can save a great deal of effort and get much better results by using the right techniques.

Quick Reference for Common Tuning Tips

There is too much detail in this paper to be quickly absorbed by the reader. Based upon my own experience there are a few recurring situations and frequently answered questions that I will list here, with references into the rest of the paper for in-depth detail. This common situation focuses primarily on servers running Solaris 2.

The First 10 Tuning Steps

1. **The system will usually have a disk bottleneck**
   In nearly every case the most serious bottleneck is an overloaded or slow disk. Use `iostat -x 30` to look for disks that are more than 30% busy and have service times of more than 50 ms. The service time is the key, this is the time between a user process issuing a read and the read completing for example so it is often in the critical path for response time. If many other processes are also accessing that disk a queue can form, and service times of over 1000 ms (not a misprint, over one second!) can easily occur as you wait to get to the front of the queue. See "Load Monitoring And Balancing" on page 102 for more details. Increasing the directory name lookup cache size can help reduce the number of disk I/Os required to manage a filesystem, see "Directory Name Lookup Cache (dnlc)" on page 45.

2. **You will be told that the system is not I/O bound**
   If you are unfamiliar with the system and are being briefed by its sysadmin you are very likely to be told that the system is *not* disk bound. Ignore this advice and insist on seeing `iostat -x 30` output for the period where the system is running sluggishly. See Step 1.

3. **After first pass tuning the system will *still* have a disk bottleneck!**
   Keep checking `iostat -x 30` as tuning progresses. When a bottleneck is removed the system may start to run faster, and as more work is done some other disk will overload. At some point you may need to stripe filesystems and tablespaces over multiple disks. See Step 1 again. (Hopefully you are getting the idea now).

4. **Poor NFS response times are hard to pin down**
   Waiting for a network mounted filesystem to respond is not counted in the same way as waiting for a local disk. The system will appear to be idle when it is really in an I/O wait state. If you know which NFS server is likely to be the problem go to it and start at Step 1 again. You should look at the

NFS operation mix with nfsstat on both the client and server and if writes are common or the servers disk is too busy configure a PrestoServe or NVSIMM in the server. See that the ethernet is not overloaded by checking that the collision rate is below one or two percent.

5. **Avoid the common memory usage misconceptions**
When you look at vmstat don't waste time worrying about where all the RAM has gone. After a while the free list will stabilize around one sixteenth of the total memory configured. The system stops bothering to reclaim memory above this level, even when you aren't running anything. See "Understanding Vmstat Output" on page 56.

6. **Don't panic when you see page-ins and page-outs in vmstat**
These are normal since all filesystem I/O is done using the paging process. Hundreds or thousands of KB paged in and paged out are not a cause for concern, just a sign that the system is working hard.

7. **Look for page scanner activity**
When you really are short of memory the scanner will be running continuously at a high rate (several hundred pages/second). If it runs in separated high level bursts you should try patching slowscan to 100 so that the bursts are made longer and slower. See "The Paging Algorithm In Solaris 2" on page 63.

8. **Look for a long run queue (vmstat procs r)**
If the run queue is more than 10 then a lot of processes are waiting for some CPU time. This waiting increases the interactive response time seen by users. Add more CPU power to the system and see "Understanding Vmstat On A Multiprocessor" on page 128.

9. **Look for processes blocked waiting for I/O (vmstat procs b)**
This is a sign of a disk bottleneck. If the number of processes blocked approaches or exceeds the number in the run queue see Step 1 again.

10. **Look for CPU system time dominating user time**
If there is more system time than there is user time and the machine is not an NFS server you may have a problem. Try to find out the source of system calls and interrupts. See "Tracing In Solaris 2" on page 32, "Monitoring The System With Proctool" on page 76 and "Kernel Profiling Using Kgmon In Solaris 2" on page 74.

Source Code

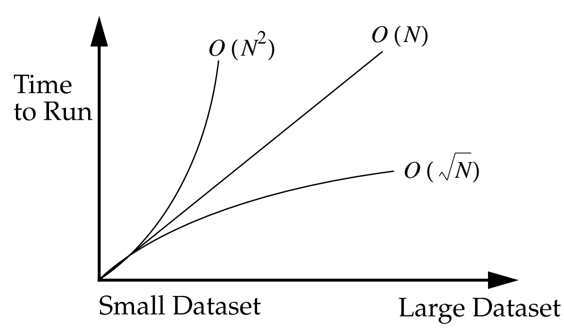
This chapter is concerned with the aspects of a system that programmers specifying and writing the software can control to affect its performance.

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 Time taken to run vs. dataset size for different classes of algorithms



1. Algorithms by Sedgewick is one to check out.

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 (the Sun3/60 has no data cache, see "A Problem With Linked Lists" on page 82) 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 miss the cache more often or page, and the cache or disk accesses can 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 have 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[1].

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 optimization 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 to subroutines by reference (i.e. passing an address rather than the number itself). This is more efficient, especially on processors like SPARC that have separate integer and floating point registers, and pass variables 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.

1. "High Performance Computing, Keith Dowd" covers these issues very well.

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. In most cases C could be used; but sloppy coding and programmers who take shortcuts make the resulting systems very hard to debug, and 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.

The execution profile of a program can be obtained using the prof or gprof tools provided with the system. One problem with these tools is that they measure the total execution from start to finish, which is less useful for window system tools that have a large start-up time. The SunPro SPARCworks Collector and Analyzer can be used on Solaris 2 to obtain this type of profile information for specific parts of a program.

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. Sentinel produce a similar product.

1. See the tcov manual page.

2. From Pure Software Inc., and described in a paper presented at Usenix, Winter 92.

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 for programs (like SPEC benchmarks) that can use maximum optimization. Some preliminary results for SPARCompilers 3.0 and Apogee 2.2 releases show substantial performance improvements for both vendors. 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[1234]. To summarize in a sentence: clean C code with lint, turn up the optimizer most of the way, profile to find hot spots and turn up the optimizer to maximum for those parts only. Look at the code in the hot spots to see if the algorithm is efficient.

Automatic Parallelization

With the introduction and shipment of a large number of multiprocessor machines it becomes cost-effective to recompile applications to utilize several processors on a single unix process. SunPro have an automatic parallelizing optimizer under development as an option for SPARCompilers 3.0. Its first release will support Fortran77, with support for Fortran 90 and C++ in later releases. Apogee ship the Kuck and Associates KAP preprocessor as an optional part of their product. This can also be used to parallelize code, although it uses Unix processes as its unit of concurrency, whereas the SunPro optimizer uses lightweight processes (Solaris 2 LWP's) in a single Unix context.

1. "Performance Tuning an Application", supplied with SunPro C and Fortran.
2. "You and Your Compiler, by Keith Bierman".
3. "SPARC Compiler Optimisation Technology Technical White Paper".
4. "High Performance Computing, Keith Dowd".

Applications vary, and some parallelize better than others. One source of examples is the individual SPECfp92 benchmarks and the recently proposed PAR93 parallel benchmark suite. See the “SPARCserver and SPARCcenter Performance Brief” and “SPARCstation 10 Product Line Technical White Paper”.

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 and I provide some pointers to my favorites below.

Mapped files

SunOS 4.X, Solaris 2.X 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 to and from a user buffer. It also allows shared files so that more efficient use of memory is possible and inter-process communication can be performed[1]. The shared library system used for dynamic linking uses mmap as its basis.

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[2].

1. SunOS 4.1 Performance Tuning
2. 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. Link with the -lbsdmalloc option.

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

- mallocmap() prints a map of the heap to the standard output.
- mallinfo() provides statistics on malloc.

Note – 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, the interface is part of SVID but the implementation is vendor dependent.

In Solaris 2 there are three versions of malloc, System V.4 malloc, SunOS 4 malloc for backwards compatibility and BSD malloc. When linking with third party libraries that use malloc take care not to mix implementations. Attempts to malloc a zero sized section of memory are handled differently by each version.

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 may end up in being cached differently from one run of a program to the next and this can cause 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

1. malloc manual page

since the performance may be better and the results will have less variance. Production code should normally dynamically link to save memory, 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.

```
% 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. 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 available at run time regardless of which routines are used. 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 things 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 command which can be used to constrain a program that is hogging too many resources. For `csh` the command is limit, for `sh` and `ksh` the command is `ulimit`. A default set of Solaris 2 resource limits are shown in Table 2, the SunOS 4 limits are similar.

Users can increase limits up to the hard system limit. The system wide default limits can only be changed by patching the kernel directly with `adb`. See "Setting Default Limits" on page 48 for details.

Table 2 Resource Limits

Resource name	Soft User Limit	Hard System Limit
cputime	unlimited	unlimited
filesize	unlimited	unlimited
datasize	524280 Kbytes	524280 Kbytes
stacksize	8192 Kbytes	261120 Kbytes
coredumpsize	unlimited	unlimited
descriptors	64	1024
memorysize	unlimited	unlimited

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
```

The maximum number of descriptors in SunOS 4.X is 256. This was increased to 1024 in Solaris 2, although the standard I/O package still only handles 256. The standards compliant definition of FILE in `/usr/include/stdio.h` only has a single byte to record the underlying file descriptor index.

If a process exceeds its memory usage limit then it is more likely to have pages taken from it when the system runs short of memory, and may be swapped out earlier than processes that are within their limits. The *memoryuse* limit does not actually prevent a process from exceeding the limit.

Tracing In SunOS 4.X

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` and has more functionality.

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
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

Table 3 `Trace` Output with Comments

Trace Output	Comments
close (1) = 0	Close stdout
close (2) = 0	Close stderr
exit (0) = ?	Exit program
%	

Tracing In Solaris 2

The `truss` command has many useful features not found in the SunOS 4 `trace` command. It can trace child processes and it can count and time system calls and signals. Other options allow named system calls to be excluded or focussed on, and data structures can be printed out in full. Here is an excerpt showing a fragment of `truss` output with the `–v` option to set verbose mode for data structures, and an example of `truss -c` showing the system call counts.

```
% truss -v all cp NewDocument Tuning
execve("/usr/bin/cp", 0xEFFFFB28, 0xEFFFFB38)  argc = 3
open("/usr/lib/libintl.so.1", O_RDONLY, 035737561304) = 3
mmap(0x00000000, 4096, PROT_READ, MAP_SHARED, 3, 0) = 0xEF7B0000
fstat(3, 0xEFFFF768)= 0
    d=0x0080001E i=29585 m=0100755 l=1  u=2     g=2     sz=14512
    at = Apr 27 11:30:14 PDT 1993  [ 735935414 ]
    mt = Mar 12 18:35:36 PST 1993  [ 731990136 ]
    ct = Mar 29 11:49:11 PST 1993  [ 733434551 ]
    bsz=8192  blks=30    fs=ufs
....
```

```
% truss -c cp NewDocument Tuning
syscall      seconds   calls  errors
_exit            .00       1
write            .00       1
open             .00      10       4
close            .01       7
creat            .01       1
chmod            .01       1
stat             .02       2       1
lseek            .00       1
fstat            .00       4
execve           .00       1
```

```
% truss -c cp NewDocument Tuning
mmap                .01      18
munmap              .00       9
memcntl             .01       1
                   ----     ---     ---
sys totals:         .07      57       5
usr time:           .02
elapsed:            .43
```

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[1]. 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. Solaris 2 has a `timex` command which provides much extended functionality. See the manual pages for more details.

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.

1. CPU percentages account for all the processors in an system so 100% represents every processor totally busy.

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[1]. 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 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 in SunOS 4.X is to uncomment 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 on by 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`. The tmpfs filesystem limits itself to prevent using up all of the swap space on a system.

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

1. The *fat* fast filesystem supports more inodes per filesystem then the regular BSD FFS.

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.

Cachefs

New in Solaris 2.3 is the cachefs filesystem type. It uses a fast filesystem to overlay accesses to a slower filesystem. The most useful way to use cachefs is to mount NFS filesystems that are mostly read only via a local UFS disk cache. The first time a file is accessed it is copied to the local UFS disk. Subsequent accesses check the NFS attributes to see if the file has changed, and if not the local disk is used. Any writes to the cachefs filesystem are written through to the underlying files by default, although there are several options that can be used in special cases for better performance.

When there is a central server that holds application binaries these can be cached on demand at client workstations. This reduces the server and network load and improves response times. See the `cfsadmin` manual page for more details. There are no monitoring tools or ways to report cache hit rate measures in this first release, but it seems to work well.

Caution – Cachefs should not be used to cache shared NFS mounted mail directories, and can slow down access to write intensive home directories.

Tuning Filesystems

Tunefs

The filesystem layout parameters can be modified using `tunefs`[1]. By default these parameters are set to provide maximum overall throughput for all combinations of read, write and update operations in both random and sequential access patterns.

Eagle DiskPak

A product from Eagle Software Inc. (phone 913-823-7257 in the USA) called DiskPak has some novel features that can improve throughput for heavily used filesystems. The product reorganizes the layout of data blocks for each file on the disk to make all files sequential and contiguous, and to optimize the placement of the UFS partial block fragments that occur at the end of files. It also has a filesystem browser utility that gives a visual representation of the block layout and free space distribution of a filesystem. The most useful capability of this product is that it can sort files based upon several criteria, to minimize disk seek time. The main criteria are access time, modification time and size. If a subset of the files are accessed most often then it helps to group them together on the disk. Sorting by size helps get a few large files separated from more commonly accessed small files. According to the vendor speedups of 20% have been measured for a mixed workload. DiskPak is available for both SunOS 4.X and Solaris 2.X.

1. See the manual page and "The Design And Implementation Of The 4.3BSD UNIX Operating System, Leffler, McKusick, Karels and Quarterman" for details of the filesystem implementation.

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. This chapter concentrates on databases in particular.

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 with SunOS 4.X

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 4; DBE 1.2 for SunOS 4.1.2 and DBE 1.3 for SunOS 4.1.3. It is sold at a very low cost 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 Oracle installations 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 at least 20% 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. Both DBE 1.3 and Solaris 2[1] implement a feature called *intimate shared memory* where the virtual address mappings are shared as well as the physical memory pages. ISM makes virtual memory operations and context switching more efficient when very large shared memory areas are used. In Solaris 2 ISM is enabled by the application when it attaches to the shared memory region. Oracle 7 and Sybase System 10 both enable ISM by setting the SHM_SHARE_MMU flag in the `shmat(2)` call.

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

1. Implemented in Solaris 2.2 and later releases.

parts of the disk must be written to maintain the filesystem, while only one block needs to be written on a raw partition. Improvements in the range 10-25% or more in database performance, and reductions in RAM requirements, have been reported when moving from filesystems to raw partitions. The prestoserve synchronous write accelerator (usually used with NFS servers) can be used with databases that have to use the filesystem and can be used as a database log file accelerator.

Database backups can be performed on small databases by copying the data from the raw partition to a filesystem. Often it is important to have a short downtime for database backups, and a disk to disk transfer is much faster than a backup to tape. Compressing the data as it is copied can save on disk space but is very CPU intensive, I would recommend compressing the data if you have a high end multiprocessor machine. E.g.

```
# dd if=/dev/rsd1d bs=56k | compress > /home/data/dump_rsd1d.Z
```

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 and temporary tablespace on their own disks 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. One approach that can be used to balance I/O over a larger number is disks is to stripe them together using Online: DiskSuite. Also see "Load Monitoring And Balancing" on page 102.

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 an option of 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, 535MB, 1.05GB, 1.3GB and 2.1GB 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 98 for more details and an explanation.

The Effect Of Indices

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 recompiled after tweaking a parameter file to increase table sizes but in Solaris 2 there is no need to recompile the kernel, it is modified by changing /etc/system and re-booting. The kernel algorithms have changed in many places between SunOS 4.X and Solaris 2. The differences are noted as SVR4 changes if they are generic and as Solaris 2 where Solaris 2 is different from generic SVR4.

Later releases of SunOS 4.X have Solaris 1.X names which I have avoided for clarity. The kernel part of Solaris 2.X is known as SunOS 5.X but this name is not often 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 recommended for each release varies as shown below.

Table 4 Tuning SunOS Releases

Release	**Extra Tuning Required (apart from maxusers)**
SunOS 4.1/Solaris 1.0	Add PMEGS patch tape or DBE-1.0, set handspread
SunOS 4.1.1/Solaris 1.0.1	Add DBE-1.1, increase buffer cache
SunOS 4.1.2/Solaris 1.0.2	Add DBE-1.2, add I/O patch 100575-02, tune pager
SunOS 4.1.3/Solaris 1.1	Add DBE-1.3, Increase maxslp, tune pager
SunOS 5.0/Solaris 2.0	Tune pager, check /etc/TIMEZONE, upgrade to 2.3!
SunOS 5.1/Solaris 2.1	Tune pager
SunOS 5.2/Solaris 2.2	Tune pager, no need to tune maxusers on small machines.
SunOS 5.3/Solaris 2.3	Tune pager, no need to tune maxusers

Solaris 2 Performance Tuning Manual

There is a manual section called "Administering Security, Performance and Accounting in Solaris 2.2". Read it, but beware that it contains a few typos and errors. The manual was written for Solaris 2.0 and some changes in the Solaris 2.2 and 2.3 kernel have obsoleted parts of the manual.

Maxusers In SunOS 4.X

The maxusers parameter is set in the kernel configuration file and a replacement kernel is compiled and linked using the new value. Many parameters and kernel tables are derived from maxusers. 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.

Using /etc/system To Modify Kernel Variables In Solaris 2

`/etc/system` is read by the kernel at start-up. It configures the search path for loadable kernel modules and allows kernel variables to be set. See the manual page for `system(4)` for the full syntax.

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 break your system. If your machine will not boot and you suspect a problem with `/etc/system` the `boot -a` option can be used. With this option the system prompts (with defaults) for its boot parameters. One of these is the configuration file `/etc/system`. Either enter the name of a backup copy of the original `/etc/system` file or enter `/dev/null`. The file should be fixed and the machine should be rebooted immediately to check that it is OK.

Maxusers In Solaris 2

The effect of maxusers has not changed but it now defaults to 8 in Solaris 2.0 and 2.1, and it is dynamically sized in Solaris 2.2 and 2.3 based upon the amount of RAM configured in the system. It is modified by placing a command in /etc/system e.g.

```
set maxusers = 200
```

Solaris 2.2

The automatic configuration of maxusers in Solaris 2.2 is based upon the value of *physmem* which is the amount of memory (in pages) after the kernel has allocated its own code and initial data space of around two MB. The automatic scaling stops when memory exceeds 128MB. For systems with 256MB or more of memory either set maxusers to the generic safe maximum of 225, or leave it and set the individual kernel resources directly. The actual safe maximum level is hardware dependent, and varies depending upon the kernel architecture.

Table 5 Maxusers Settings In Solaris 2.2

RAM Configuration	Maxusers	Processes	Name Cache
Up to and including 16 MB	8	138	226
Up to and including 32 MB	32	522	634
Up to and including 64 MB	40	650	770
Up to and including 128MB	64	1034	1178
Over 128 MB	128	2058	2266

Solaris 2.3

The maxusers setting in Solaris 2.3 is automatically set equal to the number of MB of RAM configured into the system (actually it is based upon *physmem* which does not include the two MB or so that the kernel uses at boot time). The minimum limit is 8 and the maximum automatic limit is 1024, corresponding to systems with 1GB or more of RAM. It can still be set manually in `/etc/system` but the manual setting is checked and limited to a maximum of 2048. This is a safe level on all kernel architectures, but uses a large amount of kernel memory.

Derived Parameters

Table 6 Default Settings for Kernel Parameters

Kernel Resource	Variable	Default Setting
Callout	ncallout	16 + max_nprocs[1]
Inode	ufs_ninode	max_nprocs + 16 + maxusers + 64
Name Cache	ncsize	max_nprocs + 16 + maxusers + 64
Process	max_nprocs	10 + 16 * maxusers
Quota Table	ndquot	(maxusers * NMOUNT)/4 + max_nprocs
User Process	maxuprc	max_nprocs - 5

1. ncallout no longer exists in Solaris 2.2 and later releases. The callout queue is now dynamically sized as required.

These calculations are the same in both SunOS 4 and Solaris 2. The inode and name cache variables are described in more detail later in this chapter. The other variables are not performance related.

Using Pstat In SunOS 4.X To Examine Table Sizes

The occupancy and size of some of the 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 files                     The system wide open file table
166/320 inodes                    The inode cache
 48/138 processes                 The system wide process table
13948/31756 swap                  Kilobytes 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.

Using Sar In Solaris 2 To Examine Table Sizes And Kernel Memory Allocation

Sar likes to average sizes over time, so *sar -v 1* tells sar to make one measure over a one second period. The proc-sz, inod-sz and file-sz fields are reporting the same thing as pstat -T. The file table is no longer a fixed size data structure in Solaris 2 so its size is given as zero.

```
% sar -v 1

SunOS hostname 5.2 Generic sun4c    08/05/93

00:17:08  proc-sz    ov  inod-sz    ov  file-sz    ov   lock-sz
00:17:09   44/522     0  676/634     0  292/0       0    2/0
```

The kernel dynamically allocates memory from the global free list as it needs it. Allocations of 256 bytes or less are made from a small block pool, allocations of 512 bytes to 2KB are made from a large block pool (allocations are rounded up to a power of 2). The total pool sizes never decrease, although the amount allocated will fluctuate. Oversize allocations of 4KB or more are made by allocating pages directly, and these are freed back for general use when no longer required. On high end MP machines with a lot of processes hardware and RAM to keep track of it is not unusual for the kernel to use several 10s of MB. Solaris 2.3 seems to use less than 2.2.

```
% sar -k 1

SunOS hostname 5.2 Generic sun4c    08/05/93

00:35:00 sml_mem   alloc  fail  lg_mem   alloc fail ovsz_alloc fail
00:35:01  476672  467696     0 1146880 1102848    0    3641344    0
```

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 to be cached are reported as well. A cache miss means that a disk I/O may be needed to read the directory when traversing the

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

pathname components to get to a file. A hit rate of much less than 90% will need attention. The inode cache should be at least as big as the DNLC cache. The only limit to the size of the DNLC cache is available kernel memory. For NFS server benchmarks it has been set as high as 16000 and for maxusers = 2048 it would be set at 34906.

```
% vmstat -s
... lines omitted
    79062 total name lookups (cache hits 94%)
       16 toolong
```

Inode Cache

A memory resident inode is used whenever an operation is performed on an entity in the filesystem. The inode read from disk is cached in case it is needed again and the maximum number of active and inactive inodes that the system will cache is set by *ufs_ninode*. The inodes are kept on a linked list, rather than a fixed size table. It is entirely possible that the number of open files in the system can cause the number of active inodes to exceed the limit, raising the limit allows *inactive* inodes to be cached in case they are needed again. Every entry in the DNLC cache points to an entry in the inode cache so both caches should be sized together. Since it is just a limit, *ufs_ninode* can be tweaked with adb on a running system with immediate effect. The only upper limit is the amount of kernel memory used by the inodes. The tested upper limit corresponds to `maxusers = 2048` which is the same as `ncsize` at 34906. Use `sar -k` to report the size of the kernel memory allocation, each inode uses 512 bytes of kernel memory from the lg_mem pool.

One customer written 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 on a SPARCstation 1. Since the benchmark was claimed to be representative of common disk usage patterns this shows that a large inode cache can help file intensive programs significantly.

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, indirect block and cylinder group related disk I/O only.

In Solaris 2 nbuf is used to keep track of how many page sized buffers have been allocated, and a new variable called p_nbuf (100) defines how many new buffers are allocated in one go. A variable called bufhwm controls the maximum amount of memory allocated to the buffer and is specified in KB. By default up to 2% of system memory is used, this can be increased up to 20%, and it will usually need to be increased to 10% for a dedicated NFS file server.

In Solaris 2 the buffer cache can be monitored using `sar -b`, this reports a read and a write hit rate for the buffer cache. According to "Administering Security, Performance and Accounting in Solaris 2.2" the buffer cache size (`bufhwm`) should be increased if there is a significant (say more than 50) number of reads and writes per second and the read hit rate falls below 90% or the write hit rate falls below 65%.

```
# sar -b 5 10

SunOS hostname 5.2 Generic sun4c    08/06/93

23:43:39 bread/s lread/s %rcache bwrit/s lwrit/s %wcache pread/s pwrit/s
...
Average        0      25     100       3      22      88       0       0
```

An alternative look at the buffer cache hit rate can be calculated from part of the output of `netstat -k`.

```
% netstat -k
...
biostats:
buffer cache lookups 9705 buffer cache hits 9285 new buffer
requests 0
waits for buffer allocs 0 buffers locked by someone 3 duplicate
buffers found 0
```

Comparing buffer cache hits with lookups 9285/9705 = 96% hit rate since reboot which seems OK.

Warning – netstat -k is an undocumented and unsupported option. The output is an almost raw dump of the undocumented kernel statistics interface which changes in each release of Solaris 2. Do not depend upon it.

Setting Default Limits

There is often a need to change the default hard and soft limit values. Sometimes the maximums need to be reduced to keep processes from hogging the system and sometimes the default number of files needs to be increased. Rather than setting limits in every user login script the global default limits in the kernel can be changed. In generic Unix System V Release 4 this is performed by setting some symbolic variables and rebuilding the kernel. In Solaris 2 the `/etc/system` method cannot be used since there are no variables corresponding to the individual limits, just a single `rlimits` data structure. The kernel must be patched using `adb`. The elements are laid out as shown by the `sysdef -i` command, which shows the values in hexadecimal.

```
    Soft:Hard                 Resource          byte offset in hex
Infinity:Infinity             cpu time                0: 4
Infinity:Infinity             file size               8: c
1fefe000:1fefe000             heap size              10:14
  800000: ff00000             stack size             18:1c
Infinity:Infinity             core file size         20:24
      40:      400            file descriptors       28:2c
Infinity:Infinity             mapped memory          30:34
```

To increase the default number of file descriptors per process rlimits+28 must be patched with adb as shown below.

The commands shown below permanently patch the default number of file descriptors to 128 (0x80) in the `/kernel/unix` file using "?" and also patch the current copy in memory using "/".

Warning – using adb to patch the kernel directly is a potentially dangerous operation. Mistyping a command could crash the system or render it unbootable. Save an unpatched kernel copy. Do not increase the hard limits.

```
# cp /kernel/unix /kernel/unix.orig
# adb -k -w /kernel/unix /dev/mem
rlimits,e?X
rlimits:
rlimits:    7fffffff        7fffffff        7fffffff        7fffffff
            1fefe000        1fefe000        800000          ff00000
            7fffffff        7fffffff        40              400
            7fffffff        7fffffff
```

```
rlimits+28?W80
rlimits+28/W80
rlimits,e?X
rlimits:
rlimits:   7fffffff        7fffffff        7fffffff        7fffffff
          1fefe000        1fefe000        800000           ff00000
           7fffffff        7fffffff        80                400
           7fffffff        7fffffff
```

Solaris 2 Performance Improvements

The initial developers release of Solaris 2.0 is slower than SunOS 4.1.3 on most measures. Solaris 2.1 performance improved substantially, particularly for interactive desktop use. Solaris 2.2 had further improvements, with an emphasis on database performance, I/O throughput and multiprocessor scalability. Solaris 2.3 benefits from another six months of performance tuning and bug fixes, with an emphasis on network throughput, multiuser performance and high end multiprocessor scalability. Very high SPEC LADDIS NFS performance numbers have been released using Solaris 2.3.

Some of the changes that improve performance on the latest processor types and improve high end multiprocessor scalability can cause a slight reduction in performance on earlier processor types and on uniprocessors. Trade-offs like this are carefully assessed and in most cases when part of the system is tuned an improvement must be demonstrated on all configurations.

The Solaris 2 kernel is on a diet, it will get slimmer and quicker in subsequent releases for better performance on everything from the 16MB SPARCclassic to the 20 CPU SPARCcenter 2000.

In general Solaris 2 requires more memory than SunOS 4.1.3. On small memory systems (16MB) SunOS 4.1.3 will be faster because it will be paging less[1]. On larger memory systems Solaris 2 is faster than SunOS 4.1.3 for some things and slower for others. A personal impression based upon upgrading a SPARCstation 1GX with 40MB of RAM from SunOS 4.1.2 to Solaris 2.2 is that Solaris 2.2 feels noticeably faster running OpenWindows and using Frame Maker to edit this white paper!

1. See “Solaris 2 “Standard” Tweaks” on page 71 for a way to free up some memory and help performance

Using Sar Effectively 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 interactively then go back to look at all the other measures if you need to. Sar generates average values automatically and can be used to produce averages between specified start and end times from a binary file. It is described in full in "Administering Security, Performance and Accounting in Solaris 2.2" and in "System Performance Tuning, Mike Loukides, O'Reilly".

One particularly useful facility is that the system is already set up to capture binary sar records at 20 minute intervals, and to maintain one month of past records in /var/adm/sa. This can easily be enabled by uncommenting three lines in the "sys" crontab file /var/spool/cron/crontabs/sys, and making sure that /etc/rc2.d/S21perf is enabled (the default). You will then be collecting a complete historical system load profile, so when the system is behaving "badly" the records can be compared with some previous time when the system was behaving "well". Ideally the records should be saved elsewhere for posterity on a monthly basis.

MMU Algorithms And PMEGS

A fundamental part of the kernel functionality is concerned with managing virtual and physical memory. It is useful to understand the differences between various machines in the way that virtual to physical memory translations are performed.

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.

Note that a program can still use a 500MB or more of virtual memory on these machines, it is the size of the translation cache, which affects performance, that varies. The number of PMEGS on each type of Sun is shown in Table 7.

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 64Mb. 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 Center to get hold of the PMEGS patch.

Contexts

Table 7 on page 51 shows the number of hardware contexts built into each machine. A hardware context can be thought 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 already 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. The number of various flushes can be monitored using vmstat -c.

No new machines are being designed to use this type of MMU, but it represents a large part 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	Contexts	Total VM
Cypress SPARC/MBus chipsets e.g.	4 Kb	64	4096	256Kb
SPARCserver 600 -120 and -140 &	256Kb	64	4096	16Mb
Tadpole SPARCbook	16Mb	64	4096	1024Mb

Table 8 SPARC Reference MMU Characteristics

Processor Types	Page Sizes	TLB	Contexts	Total VM
Texas Instruments SuperSPARC e.g.	4 Kb	64	65536	256Kb
SPARCserver 600 -41, -52 and -54 &	256Kb	64	65536	16Mb
SPARCstation 10 -30, -41, -52,and -54	16Mb	64	65536	1024Mb
Fujitsu SPARClite embedded CPU	4 Kb	32	256	128Kb
	256Kb	32	256	8Mb
	16Mb	32	256	512Mb
Texas Instruments MicroSPARC e.g.	4Kb	32	64	128Kb
SPARCclassic and SPARCstation LX	256Kb	32	64	8Mb
	16Mb	32	64	512Mb

A detailed description of the hardware involved can be found in "Multiprocessor System Architectures, Ben Catazaro, SunSoft Press".

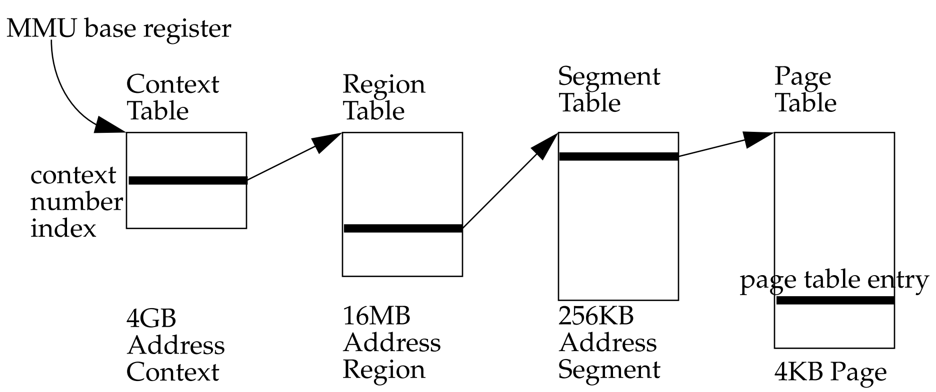
There are four current implementations, the Cypress uniprocessor 604 and multiprocessor 605 MMU chips, the MMU that is integrated into the SuperSPARC chip, the Fujitsu SPARClite and the highly integrated MicroSPARC.

Unlike the sun4 MMU 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 are being optimized to use the 256Kb or 16Mb pages in certain cases, for mapping frame buffers (in Solaris 2.3) and parts of the kernel. This requires contiguous and aligned physical memory for each mapping, which is hard to allocate except for special 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 SPARC Reference MMU Table Walk Operation

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 using software. 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 MMU then does a table walk by indexing and following linked lists to find the right page translation to load into the TLB. This is shown in Figure 1. The table walk is optimized by the MMU hardware, which keeps the last accessed context, region and segment values in registers so that the only operation needed is to index into the page table with the address supplied and load a page table entry. For the larger page sizes the table walk stops with a special PTE at the region or segment level. The size of the page translation 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. Both DBE 1.2 and SunOS 4.1.3 set `npts` much higher and it does not normally need to be tweaked.

Figure 1 SPARC Reference MMU Table Walk



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 of simultaneously mapped virtual memory. The SRMMU is reloading continuously as the CPU uses more than this small set of pages but it has an exceptionally fast reload so there is a low overhead.

The Paging Algorithm and How To Tune It

The paging algorithm is used to manage and allocate physical memory pages to hold the active parts of the virtual address space of processes running on the system. First the monitoring statistics are explained; then the algorithm is explained; and finally recommended changes in the tuning parameters for different circumstances are made.

Understanding Vmstat Output

The paging activity on a Sun can be monitored using vmstat or sar. Sar is better for logging the information, but vmstat is more concise and crams more information into each line of text for interactive use. SunOS 4.X output is shown here. Solaris 2.0 uses the old 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 does not explain how to interpret the information so clarification of some fields is in order.

Table 9 Vmstat fields explained

Field	Explanation
procs r	Processes in run queue, different semantics in SunOS4.X and Solaris 2[1]
procs b	Processes blocked for resources, paging, I/O etc
procs w	Processes runnable but swapped out
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
fre	Free list memory in Kbytes, the pages of RAM that are ready to be reused
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, avoiding I/O (previously discarded)
at	attaches to pages already in use by other processes (ref count incremented)
pi	kilobytes per second paged in causing disk or network reads
po	kilobytes per second paged out due to memory shortage or sync/fsflush
fr	kilobytes freed per second by the scanner
de	artificial memory deficit set at swap out to prevent immediate swapin
sr	pages scanned by clock algorithm, per-second

1. See “Vmstat Run Queue Differences In Solaris 2” on page 132

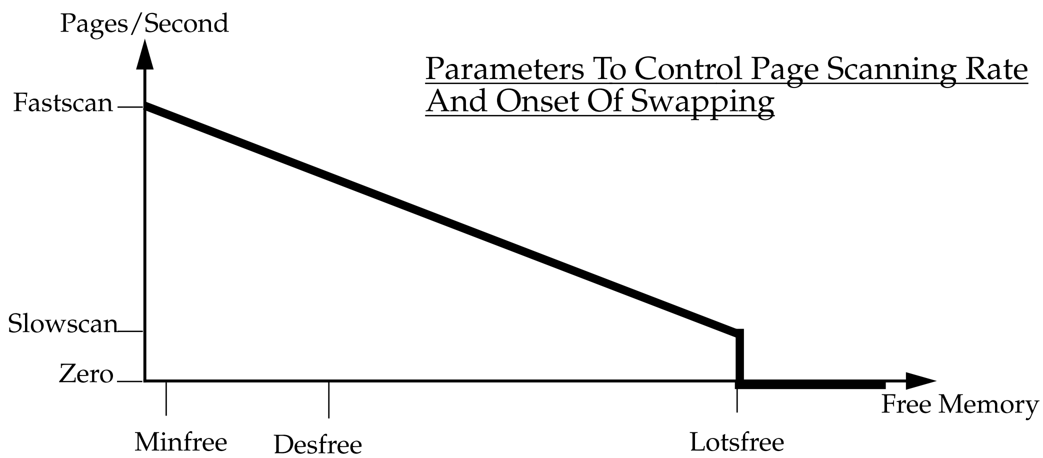
Looking at the above log of vmstat it can be seen that some CPU activity in the first few entries was entirely memory resident. This was followed by some paging and as the free memory dropped below 256K the scanning algorithm woke up and started to look for pages that could be reused to keep the free 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 SunOS 4 desktop machine that has been running for a while and is not idle.

The Paging Algorithm In SunOS 4.X

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 speed 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 66% of 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 In SunOS 4.X

The default values change in each OS release, and vary for each kernel architecture for the same release of SunOS 4.X.

Physmem

This is set to the number of pages of usable physical memory. Some other variables are based on physmem. Pages are 4Kbytes on most SPARC machines[1]. Adb prints out physmem in hexadecimal when it is used to patch a live system.

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 usually set to 8 pages and clamped at desfree/2. The SunOS 4 sun4m kernel sets it to 128 pages.

1. See Table 7 on page 51 for a full list.

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 usually set to 25 pages and is clamped to (physmem/16). The SunOS 4 sun4m kernel sets the value to 256 pages.

lotsfree

This is the memory limit that triggers the page daemon to start working if free memory drops below it. It is usually set to 64 pages and is clamped at (physmem/8). The SunOS 4 sun4m kernel sets it to 512 pages.

fastscan

This is the number of pages scanned per second by the algorithm when there is `minfree` available and it is usually set to 1000. There is a linear ramp up from `slowscan` to `fastscan` as free memory goes from `lotsfree` to zero. The SunOS 4 sun4m kernel sets it to (physmem/2).

slowscan

This is the number of pages scanned per second by the algorithm when there is just under `lotsfree` available and it is usually set to (`fastscan`/10). There is a linear ramp up from `slowscan` to `fastscan` as free memory goes from `lotsfree` to zero.

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 peak 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 15 on page 97 for disk specifications.

handspread

Handspread is set to (physmem*pagesize/4), but is increased during initialization to be at least as large as fastscan which makes it (physmem*pagesize/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.

Warning – using adb to patch the kernel directly is a potentially dangerous operation. Mistyping a command could crash the system or render it unbootable. Save an unpatched kernel copy.

```
# cp /vmunix /vmunix.orig
# 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 a stop/go effect can occur where the scanning algorithm will be triggered by a shortage of memory and it will free almost every page it finds until there is a lot 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 (SunOS 4.X)

Sleep Time Based Swap Outs

If a process has been sleeping for more than 20 seconds then it is very likely to be swapped out even if there is a lot of free memory available. 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). 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. These operations are also known as *soft swaps* and are not included in the swapouts reported by vmstat -S. In practice only the private pages for the process are swapped out, and these pages are put on the free list. When the swapin occurs the pages may still be on the free list so can be reclaimed without any disk reads.

Memory Shortage Based Swap Outs

The SunOS 4 kernel keeps track of the ten biggest processes according to their resident set size (RSS). When there is a memory shortage (see the description of minfree and desfree above) 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. This is also known as a *hard swap* and is reported by vmstat -S as "so" in swapouts per second. It is very rare for memory to become scarce enough for this type of swap out to occur.

Swap Ins

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.

The Paging Algorithm In Solaris 2

The basic algorithm is similar to SunOS 4.X, but the details have changed sufficiently for the tuning parameters to have different meanings in some cases (although they have the same names). This part of Solaris 2 has some tuning variables derived from System V, and some from BSD4.3 via SunOS 4, but the algorithm now uses Solaris 2 kernel threads and works alongside the scheduler. Swapping is now a scheduler function, and Solaris 2.1 and 2.2 machines do not perform any swapping in normal operation. Solaris 2.3 swaps in some circumstances. Pageouts are now queued and clustered so that the random pageouts are reorganized into large sequential writes. This removes one of the primary reasons why swapping was needed in SunOS 4.X.

Swap Space

One other change is that swap space on disk is no longer required to backup physical RAM. The total allocatable memory in SunOS 4.X is equal to the swap space, and large memory machines require an even larger swap space to backup the RAM. In Solaris 2 if you have enough RAM you do not need to configure any swap space at all, so pageouts cannot occur! If you ran on SunOS 4.X with 30-40MB of swap space, then you could configure Solaris 2 with 48MB of RAM and no swap disk. The page scanning algorithm still needs to find unused pages to put on the free list, but it cannot cause pageouts.

Tuning Parameters

The tuning parameters are described below.

Caution – The kernel algorithms are subject to change and the algorithms described are only valid for Solaris 2.1, 2.2 and 2.3. Solaris 2.4 is very likely to have changes to both the algorithms and the default values.

Physmem

This is set to the number of pages of usable physical memory. As previously mentioned, the maxusers calculation is based upon physmem in Solaris 2.2 and 2.3. If you are investigating system performance and want to run tests with reduced memory on a system, you can set physmem in /etc/system and reboot to prevent a machine from using all its RAM. The unused memory still uses some kernel resources, and the page scanner still scans it, so if the reduced memory system does a lot of paging the effect may not be the same as physically removing the RAM.

minfree

Minfree is set to (physmem/64). The kernel uses minfree for the following purposes:

- When exec'ing a small program (under 280K which is set by pgthresh), the entire program is loaded in one go rather than being paged in piecemeal as long as doing so would not reduce freemem below minfree.
- During a copy-on-write operation, the original page may be stolen to provide the copy if freemem is less than minfree.
- I/O Readahead clustering is disabled if freemem is less then minfree.
- The scheduler will not swap in processes while there is less than minfree + tune_t_gpgslo available. This is a very rare condition.

desfree

Desfree is set to (physmem/32). The kernel uses desfree for the following purposes:

- At the point where pages are taken from the free list, if freemem is less than desfree an immediate wakeup call is sent to the pageout daemon, rather than waiting for pagedaemon to wake up on its own (which happens four times per second). This is similar to SunOS 4.
- The number of entries in the queue of pending pageout I/Os is set to desfree (which is sized in pages).

lotsfree

Lotsfree is set to (physmem/16). The kernel uses lotsfree for the following purposes:

- During the 100Hz clock routine a test is made four times a second to see if freemem is less than lotsfree. If so a wakeup is sent to the pageout daemon.
- Lotsfree is the baseline for the scan rate interpolation. When freemem is the same as lotsfree then the scan rate is set to slowscan.
- If freemem is less than lotsfree the kernel tries to free some transitory page sized buffers rather than holding onto them for future use.

fastscan

Fastscan is set to (physmem/2). It is used in the scan rate interpolation as the notional scan rate when freemem is zero.

slowscan

This is the number of pages scanned per second by the algorithm when there is `exactly lotsfree` available and it is set to (fastscan/10). There is a linear ramp up from `slowscan` to `fastscan` as free memory goes from `lotsfree` to zero.

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 15 on page 97 for disk specifications. The value is divided by four during system initialization since the pageout daemon runs four times per second and the resulting value is the limit on the number of pageouts that the pageout daemon will add to the pageout queue in each invocation. Note that in addition to the clock based invocations, an additional invocation will occur whenever more memory is allocated and freemem is less than desfree, so more than maxpgio pages will be queued per second when a lot of memory is allocated repeatedly

Note – Changes to maxpgio only take effect after a reboot, so it cannot be tweaked on a running system.

handspread

Handspread is set to (physmem*pagesize/4), but is increased during initialization to be at least as big as fastscan which makes it (physmem*pagesize/2).

tune_t_ gpgslo

This variable is a Unix System V.3 derived feature. In Solaris 2 it defaults to 25 pages and is the threshold used by the scheduler to decide whether to begin swapping out processes. It is almost impossible to get the free list down below 25 pages so this is a very rare condition.

autoup and tune_t_fsflushr

Unlike SunOS 4.X where the update process does a full sync of memory to disk every 30 seconds, Solaris 2 uses the fsflush daemon to spread the sync workload out. Autoup is set to 30 seconds by default, and this is the maximum age of any memory resident filesystem pages that have been modified. Unlike update, fsflush wakes up every 5 seconds (set by tune_t_fsflushr) and checks a portion of memory on each invocation (5/30= one sixth of total RAM by default). The pages are queued on the same list that the pageout daemon uses and are formed into clustered sequential writes. The system counts these writes as pageouts.

max_page_get

This variable is set to half the number of pages in the system and limits the maximum number of pages that can be allocated in a single operation. In some circumstances a machine may be sized to run a single very large program that has a data area or single malloc space of more than half the total RAM. It will be necessary to increase max_page_get in that circumstance.

Warning – If max_page_get is increased too far and reaches total_pages (a little less than physmem) then deadlock can occur and the system will hang trying to allocate more pages than exist.

Swapping Out And How To Get Back In (Solaris 2)

Under extreme artificially induced conditions a Solaris 2.2 system was made so short of memory that it was observed to swap out a process, but for all practical purposes swapping can be ignored in Solaris 2.1 or 2.2. The time based soft swapouts that occur in SunOS 4.X are no longer implemented. Vmstat -s will report total numbers of swapins and swapouts, which are always zero. In Solaris 2.3 the code was changed so that prolonged memory shortages can trigger swapouts of inactive processes.

Sensible Tweaking

The default settings for SunOS 4.X were derived from the BSD4.3 values designed 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 were overhauled for Solaris 2.1 where they were optimized for good overall window system performance with 16MB. No changes were made to the defaults for Solaris 2.2 or 2.3. Changes are expected for Solaris 2.4.

Note – The suggestions below are a mixture of theory, informed guesswork and trial and error testing,. There are no right answers, as the ideal parameters depend upon the application workload. The default parameters do not scale ideally for use on very large memory systems.

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 size of the free list to cope with frequent and large memory allocation demands. A pure database server, compute server or single application document publishing or CAD workstation may start-up then stay very stable, with a long time between process start-ups. This type of system can use a reduced size free list, as the pages are better utilized by the application rather than sitting idly in the free list.

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. When the memory has been found the process continues and the situation can repeat several times. These interruptions in process start-up time are 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 weed out 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 512 pages or 2048Kb. 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. Solaris 2 sets lotsfree to (physmem/16) which results in an improvement in start-up times, but a large increase in paging rates when 16MB machine is upgraded from SunOS 4. I recommend setting the `lotsfree` parameter to (physmem)/32 on all machines as a compromise. `Desfree` is usually set to half of lotsfree, so I recommend (physmem/64). You can try out tweaks 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.

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

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 (physmem/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 400 on a 32Mb machine and about 12500 on a 1Gb machine). Solaris 2.0 used fixed scan rates, but Solaris 2.1, 2.2 and 2.3 use the same scaled rates described above for all machines. These very high scaled scan rates do not allow enough memory references to occur, so all but the most active pages will be put on the free list, and unnecessary pageouts may occur. The pageout daemon may also consume excessive system CPU time. `Slowscan` should be set quite low at a fixed, unscaled level of perhaps 100 pages/second so that the number of pages scanned at the slowest rate doesn't immediately cause a paging I/O overload. `Fastscan` should be halved to (physmem/4), which will automatically remove the override that currently forces the handspread value up from 90˚ to 180˚. The ramp up from slowscan to fastscan as free memory drops will be steep so there is no need to increase slowscan as memory is added.

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 return to 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 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.

Warning – These values should be scaled according to the configuration of the machine being tuned. Save an unpatched copy of the kernel and be very careful when using `adb`. Errors can crash the system or render it unbootable.

```
# 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 (current
                   value cannot be tweaked)
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 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
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 "Standard" Tweaks

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

These numbers are for a 16MB machine with a relatively constant workload.

```
# adb -k -w /dev/ksyms /dev/mem
physmem ffd        physmem is the total number of physical memory
                   pages in hex
slowscan/W0x64     set slow scan rate to 100 pages/second
fastscan/W0x400    set fast scan rate to 1024 pages/second
minfree/W0x10      set minfree to 64 Kbytes
desfree/W0x40      set desfree to 256 Kbytes
lotsfree/W0x80     set current lotsfree to 512 Kbytes
```

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

```
# Settings for 16MB of RAM
set slowscan = 100     # set slow scan rate to 100 pages/second
set fastscan = 0x400   # set fast scan rate to 2048 pages/second
set lotsfree = 0x80    # set current lotsfree to 128 pages, 512KB
set maxpgio  = 0x32    # set maxpgio to 50 pages/sec
```

At the other end of the scale, on a 1 GB machine with four swap disks, and a very active workload (compiles or batch streams of small jobs), slowscan should be set to 100, fastscan should be halved, and lotsfree, desfree and minfree should be left at their defaults. Maxusers needs to be set higher explicitly if required and maxpgio should be set to 4 x 60.

```
set maxusers = 500       # increase kernel table sizes
set fastscan = 0x10000   # set fastscan rate to 65536 pages/sec
set slowscan = 100       # fix initial scan rate at 100
set maxpgio  = 240       # set maxpgio to 60 pages/sec for each
                         of four 90 rps disks
```

To decide whether the system is active, does a lot of process creation, and needs a large free list, use sar -c to monitor the number of fork and exec system calls. If there is a lot less than 1 per second averaged over a 20 minute period then the free list can be reduced in size.

Configuring Devices

Kernel Configuration In 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 and enabled in /etc/rc.local makes a good DL25 kernel for the ELC.

Kernel Configuration In 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.

Mapping Device Nicknames To Full Names In Solaris 2

Note – This is very powerful and flexible, but is not documented well in the manuals. A useful command script that does the mapping is listed below.

The boot sequence builds a tree in memory of the hardware devices present in the system, which is passed to the kernel and 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 `drvconfig tapes` and `disks` commands that allow you to configure a system manually. The file /etc/path_to_inst maps hardware addresses to symbolic device names and an extract from a simple configuration with the symbolic names added is shown below. When a large number of disks are configured it is important to know this mapping so that iostat and related commands can be related to the output from df. The sbus@1 part tells you which SBus is used (an SC2000 can have up to 10 separate SBuses) the esp@0 part tells you which SBus slot the "esp" (SCSI) controller is in. The sd@0 part tells you that this is SCSI target address 0. The /dev/dsk/c0t0d0s2 device name indicates SCSI target 0 on SCSI controller 0 and is a symbolic link to a similar hardware specification to that found in /etc/path_to_inst. The extra ":c" at the end of the name in /devices corresponds to the "s2" at the end of the name in /dev. Slice s0 is partition :a, s1 is :b, s2 is :c etc.

```
% more /etc/path_to_inst
...
"/fd@1,f7200000" 0         fd0
"/sbus@1,f8000000/esp@0,800000/sd@3,0" 3sd3
"/sbus@1,f8000000/esp@0,800000/sd@0,0" 0sd0
"/sbus@1,f8000000/esp@0,800000/sd@1,0" 1sd1
```

```
% iostat -x 5
                                  extended disk statistics
disk       r/s  w/s   Kr/s   Kw/s wait actv  svc_t  %w  %b
fd0        0.0  0.0    0.0    0.0  0.0  0.0    0.0   0   0
sd0        0.1  0.1    0.4    0.8  0.0  0.0   49.3   0   1
sd1        0.1  0.0    0.8    0.1  0.0  0.0   49.0   0   0
sd3        0.1  0.1    0.6    0.8  0.0  0.0   75.7   0   1
```

```
% df -k
Filesystem             kbytes    used   avail  capacity  Mounted on
/dev/dsk/c0t3d0s0       19107   13753    3444     80%    /           sd3
/dev/dsk/c0t3d0s6       56431   46491    4300     92%    /usr        sd3
/proc                       0       0       0      0%     /proc
fd                          0       0       0      0%     /dev/fd[1]
swap                     2140      32    2108      1%     /tmp
/dev/dsk/c0t3d0s5       19737   17643     124     99%    /opt        sd3
/dev/dsk/c0t1d0s6       95421   71221   14660     83% /usr/openwin sd1
/dev/dsk/c0t0d0s2      308619  276235    1524     99%    /export     sd0
```

1. /dev/fd is a file descriptor filesystem type, nothing to do with floppy disks!

```
% ls -l /dev/dsk/c0t0d0s2
lrwxrwxrwx  1 root          51 Jun  6 15:59 /dev/dsk/c0t0d0s2 ->
../../devices/sbus@1,f8000000/esp@0,800000/sd@0,0:c
```

The following csh/nawk script can be used to print out the device to nickname mappings.

Figure 2 whatdev - device to nickname mapping script

```
#!/bin/csh
# print out the drive name - st0 or sd0 - given the /dev entry
# first get something like "/iommu/.../.../sd@0,0"
set dev = `/bin/ls -l $1 | nawk '{ n = split($11, a, "/"); \
split(a[n],b,":"); for(i = 4; i < n; i++) printf("/%s",a[i]); \
printf("/%s\n", b[1]) }'`
if ($dev == "") exit
# then get the instance number and concatenate with the "sd"
nawk -v dev=$dev '$1 ~ dev { n = split(dev, a, "/"); split(a[n], \
b, "@"); printf("%s%s\n", b[1], $2) }' /etc/path_to_inst
```

An example of its use.

```
% df /
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/dsk/c0t3d0s0      388998  284625   65483     81%    /
% whatdev /dev/dsk/c0t3d0s0
sd3
```

Kernel Profiling Using Kgmon In Solaris 2

The kernel can be profiled in the same way that user programs can be profiled using special compiler options and gprof. The standard kernel has clock sample profiling enabled, for SunSoft's own in-house performance tuning work the kernel can be recompiled to include call graph profiling as well.

Warning – kernel profiling is an unsupported and undocumented capability. The interface and capabilities may change in each release of Solaris 2 and if the system misbehaves or crashes while profiling is enabled you are on your own.

The profile is collected using the "kgmon" utility. It allocates some profiling buffers, collects the profile and dumps the results to be processed by gprof. It must be run as root. An example sequence of commands is shown below

There is no manual page, use kgmon -- to show the options.

```
# kgmon --
Usage: kgmon [ -i -b -h -r -p -s -d ]
    -i = initialize profiling buffers
    -b = begin profiling
    -h = halt profiling
    -r = reset the profiling buffers
    -p = dump the profiling buffers
    -s = snap shot the profiling data
    -d = deallocate profiling buffers
# kgmon -i
one moment
NOTICE: Profiling kernel, textsize = 626692 [f0004000..f009d004]
NOTICE: Profiling modules, size = 5267000 [ff011000..ff516e38]
(954282 used)
NOTICE: need 139600 bytes per cpu for clock sampling
profiling has been initialized
WARNING: call graph profiling is not compiled into this kernel
WARNING: only clock sample profiling is available
# kgmon -b
kernel profiling is running
```

Run the workload at this point.

```
# kgmon -h
kernel profiling is off
# kgmon -p
dumping cpu 0 - one moment
a.out symbols for the kernel and modules have been dumped to
gmon.syms
profiling data has been dumped to gmon[n].out
# kgmon -d
profiling buffers have been deallocated
# /usr/ccs/bin/gprof gmon.syms gmon.out >kgmon.gprof
```

Since call graph profiling is not enabled the first part of the output file is not very interesting. The “flat” profile at the end is the most useful part and it is shown below. In this case the profile was collected for about 30 seconds on a SPARCstation 1 that was mostly idle and user CPU bound, with some I/O activity. On multiprocessor machines the idle loop also calls disp_getwork, which means that the idle CPU is checking the cache affinity algorithm to see if a process should be migrated off the despatch queue of a busy CPU.

```
granularity: each sample hit covers 4 byte(s) for 0.03% of 32.83
seconds
  %  cumulative    self            self    total
time   seconds   seconds  calls  ms/call  ms/call name
66.3      21.77    21.77                          idle [1]
 1.8      22.37     0.60                      vac_usrflush [2]
 1.6      22.90     0.53                          bcopy [3]
 1.6      23.43     0.53                  sunm_pmgswapptes [4]
 1.3      23.86     0.43                         _level10 [5]
 1.2      24.25     0.39                       mutex_enter [6]
 0.9      24.54     0.29                        mutex_exit [7]
 0.8      24.79     0.25                        syscall [8]
 0.6      24.99     0.20                      sunm_pmgload [9]
 0.5      25.17     0.18                        .syscall [10]
 0.5      25.34     0.17                     sunm_ptesync [11]
 0.5      25.50     0.16                        blkclr [12]
 0.5      25.65     0.15                        clock [13]
 0.5      25.80     0.15                        hmetopmg [14]
 0.4      25.94     0.14                        pgcopy [15]
 0.4      26.07     0.13                        poll [16]
 0.4      26.19     0.12                      _interrupt [18]
 0.4      26.31     0.12                        fsflush [17]
 0.3      26.42     0.11                        .div [19]
 0.3      26.53     0.11                        resume [20]
 0.3      26.64     0.11                        rw [21]
 0.3      26.75     0.11                        strpoll [22]
.... and so on
```

The kernel routine names are fairly descriptive, but access to kernel sourcecode or a Solaris System Internals course is needed to make good use of this information.

Monitoring The System With Proctool

Proctool is a freely available tool developed by Walter Nielsen and Morgan Herrington of Sun. It can be obtained via anonymous ftp on Usenet from the machine opcom.sun.ca in the directory /pub/binaries/proctool_2.2. Proctool uses features only found in Solaris 2 and it is release specific. Proctool 2.2 only works with Solaris 2.2, Proctool 2.3 for Solaris 2.3 is under development.

Proctool monitors and controls processes on a system, providing an OpenLook GUI interface to ps(1) and providing more flexibility than the public domain top utility. It extracts a huge amount of information from the data structures maintained in /proc for each process and can plot a lot of data graphically on a per system or a per process basis.

References

- "Administering Security, Performance and Accounting in Solaris 2.2"
- "Building and Debugging SunOS Kernels, by Hal Stern"
- "SPARCstation 2 Performance Brief"
- "The Design And Implementation Of The 4.3BSD UNIX Operating System, Leffler, McKusick, Karels and Quarterman"
- "SunOS System Internals Course Notes"
- "System Performance Tuning, Mike Loukides, O'Reilly"

Memory

6

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 performance problems and 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[1]. 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. SPARC processors rated at over 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

1. “High Performance Computing, Keith Dowd” covers this subject very well.

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 an initial 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 51 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

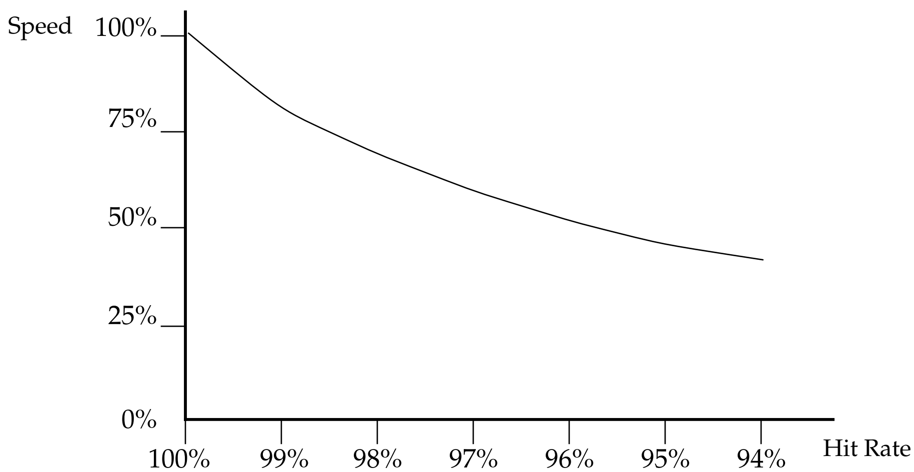
A SPARCstation 2 has its 64Kb of cache organized as 2048 blocks of 32 bytes each. When the CPU accesses an address the cache controller checks to see if the right data is in the cache, if it is then the CPU loads it without stalling. If the data needs to be fetched from main memory then the CPU clock 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 10 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



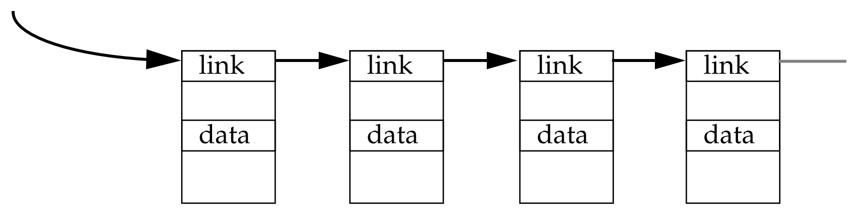
There is a dramatic increase in execution time as the hit rate drops. Although a 96% hit rate sounds quite high you can see that the program will be running at half speed. Many small benchmarks like Dhrystone run at 100% hit rate. The SPEC92 benchmark suite runs at a 99% hit rate given a 1Mbyte cache[1]. 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 19 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 then 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 91 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 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. This executes in 9 cycles on a SPARCstation 2 if it hits the cache, and 59 cycles if both loads miss.

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 in 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 “Multiprocessors” on page 119”

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 11 Virtual Write Through Cache Details

Machine	Clock	Size	Line	Read Miss	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)
SS330,370,390	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

The larger desk-side SPARCserver's from Sun use this cache type.

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 12 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	30 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. The cache coherency mechanisms required for a multiprocessor machine also introduce extra cycles. There is a difference between the number of MBus cycles taken for a cache miss (the bus occupancy) and the number of cycles that the CPU stalls for. A lower bus occupancy means that more CPU's can be used on the bus.

Physical Write Back Caches

The SPARCserver 600 and SPARCstation 10 with SuperCache, and the SPARCserver 1000 and SPARCcenter 2000 use this cache type for their second level cache. The first level cache is described later, in the section on on-chip caches.

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 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 cuts the number of cache tags required at the expense of needing an extra three valid bits in each tag. In XDBus mode for the SPARCserver 1000 the same chipset switches to two 64 byte blocks. The SPARCcenter 2000 takes advantage of this to use four 64 byte blocks per 256 byte line, for a total of 2MB of cache on a special module.

In Xbus mode the cache request and response are handled as separate transactions on the bus, and other bus traffic can interleave for better throughput but delayed response. The larger cache line takes longer to transfer, but the memory system is in multiple independent banks, accessed over two interleaved buses on the SPARCcenter 2000. When many CPUs access memory at once the single MBus memory system clogs up faster than the XDbus.

Table 13 Physical Write Back Cache Details

Machine	Size	Line (block)	Miss Cost	CPU Clock Rate
SPARCserver 600 model 41	1024Kb	128 (32) bytes	40+ cycles	40MHz
SPARCstation 10 model 51	1024Kb	128 (32) bytes	40+ cycles	50MHz
SPARCserver 1000	1024Kb	128 (64) bytes	Variable	50MHz
SPARCcenter 2000	2048Kb	256 (64) bytes	Variable	50MHz

On-Chip Caches

Example Machines

Highly integrated SPARC chipsets like the Texas Instruments SuperSPARC (Viking) used in the SPARCstation 10, MicroSPARC (Tsunami) used in the SPARCstation LX and SPARCclassic and the Fujitsu MB86930 (SPARClite) use this cache type. The Ross HyperSPARC uses a hybrid on-chip instruction cache with off chip unified cache. The latest SPARC design is the MicroSPARC II, which has four times the cache size of the original MicroSPARC and has other performance improvements. It is made by Fujitsu, who also recently bought Ross from Cypress. One other recent development is the Weitek PowerUP, which is a plug in replacement for the SS2 and IPX CPU chip that adds on-chip caches and doubles the clock rate.

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 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. If the on-chip cache is connected directly to main memory it has a 10 cycle effective miss cost, if it is used with a SuperCache it can transfer data from the SuperCache to the on-chip cache with a 5 cycle cost.

Table 14 On-Chip Cache Details

Processor	I-Size	I-line	I-Assoc	D-Size	D-line	D-Assoc	D-Miss
MB86930	2KB	16	2	2KB	16	2	?
MicroSPARC	4KB	32	1	2KB	16	1	4-9cycles
MicroSPARC II	16KB	32	1	8KB	16	1	?
PowerUP	16KB	32	1	8KB	32	1	?
HyperSPARC	8KB	32	1	256KB[1]	32	1	?
SuperSPARC	20KB	64 (32)	5	16KB	32	4	5-10cycles

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

1. "The SuperSPARC Microprocessor Technical White Paper"

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.

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. There is a mode bit to control this, and due to bugs in early versions of SuperSPARC it is disabled in Solaris 2.2 by default. It will be switchable or automatically configured in a future release.

Cache Block Prefetch

A mode bit can be toggled in the SuperCache which causes cache blocks to be fetched during idle-time on the MBus. If a cache line has invalid sub-blocks, but a valid address tag, then the missing sub-blocks will be prefetched. This mode is turned on by default in Solaris 2.2. It will be switchable in a future release since some workloads run better without it.

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.

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 a burst transfer.

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 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[2]. The SPARCserver 600 has a similar I/O cache on its VMEbus to SBus interface[3] 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, and on the XDBus in the SPARCserver 1000 and SPARCcenter 2000 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.

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

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

3. "SPARCserver 10 and SPARCserver 600 White Paper"

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
- 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];
for(i=0; i < BUFSIZE; i++)
    destination[i] = source[i];
```

The compiler will allocate both arrays adjacently in memory so they will be aligned and the copy will run very slowly. The library bcopy routine unrolls the loop to read for one cache line then write, which avoids the problem.

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 size of the block transfer is examined to see if the setup overhead is worth it, and small copies are done in software.

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 subject of windows and graphics. 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.

- “SunPHIGS / SunGKS Technical White Paper”
- “XGL Graphics Library Technical White Paper”
- “SPARCserver 1000 Performance Brief”
- “SPARCclassic X Performance Brief”
- “SPARCserver Sizing Guide for X terminals”
- “TurboGXplus Graphics Technology, A White Paper”
- “SPARCstation 10SX Graphics Technology, A White Paper”
- “SPARCstation 10ZX and SPARCstation ZX Graphics Technology, A White Paper”
- “SPARCstation ZX, SPARCstation 10ZX and SPARCstation 10 TurboGXplus Graphics Performance Brief”
- “Solaris XIL 1.0 Imaging Library White Paper”

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

Disk specifications are commonly reported using the "best case" approach which is disk format independent. 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 (Millions of bytes/s)
- The disk controller interface used (ST506, ESDI, IDE,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 kilobytes (1024) of your disk. The entry for a typical disk is shown in Figure 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 kilobytes/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 kilobytes/sec which confirms the calculated result. Some common Sun disks are listed in table 8-1 with kilobytes of 2^{10} = 1024. The data rate for ZBR drives cannot be calculated since the format.dat nsect entry is fudged.

Table 15 Disk Specifications

Disk Type	Capacity	Peak MB/s	Data Rate	RPM	Seek
Sun 0207 SCSI	203148 KB	1.6	1080 KB/s	3600	16ms
Sun 0424 ZBR SCSI	414360 KB	2.5-3.0	variable	4400	14ms
Sun 0535 ZBR SCSI	522480 KB	2.9-5.1	variable	5400	11ms
Sun 0669 SCSI	653670 KB	1.8	1620 KB/s	3600	16ms
Sun 1.3G ZBR SCSI	1336200 KB	3.25-4.5	variable	5400	11ms
Sun 1.3G ZBR IPI	1255059 KB	3.25-4.5	2610-3510 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	variable	5400	11ms
CDC 911M IPI	889980 KB	6.0	4680 KB/s	3600	15ms
Sun 2.1G ZBR DFSCSI	2077080 KB	3.8-5.0	variable	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 and higher data rates are possible. The 1.3GB drive zones mean that peak performance is obtained from the first third of the disk up to cylinder 700.

Table 16 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. The number of sectors per track is set to make sure partitions start on multiples of 16 blocks, and does not accurately reflect the geometry of the outer zone. The 1.3GB IPI drive outer zone happens to match format.dat but the other ZBR drives have more sectors than format.dat states.

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 = 173KB/s = 2 / (0.011 + (2 / 3510))

8KB data rate = 602KB/s = 8 / (0.011 + (8 / 3510))

56KB data rate = 2078KB/s = 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 as well. 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.

1. With SunOS 4.1.1 and subsequent releases of SunOS 4.X.

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[1] 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 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 multiuser 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 optimizer 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 organized as 128 - 8 kilobyte 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

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

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. The 1.3Gb IPI ZBR disk has its zone map stored in the ROM on the ISP-80 controller so that the variable geometry can be allowed for.

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.

Iostat In SunOS 4.X

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 utilization 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
```

```
% 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
```

The second form of `iostat -D` was introduced in SunOS 4.1 and continues unchanged in Solaris 2. 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.

Iostat In Solaris 2

Solaris 2 has three forms of iostat output, one is the same "-D" option described above for SunOS 4.X, the other two are described below. The default output with no options has changed to show some new values in a similar layout. For each disk the same values as before are reported for KB per second (now labelled Kps rather than bps), and transfers per second. A new value of service time in milliseconds is also reported. The service time is the average time taken to service a complete I/O request, including time spent waiting for preceding requests in the I/O queue to finish. The current measurement method tends to overestimate the actual service time, so the results are not very dependable. Nice CPU has been dropped in favor of time spent blocked waiting for I/O. Unlike SunOS 4.X floppy disk activity is reported in Solaris 2.

```
% iostat 5
      tty          fd0           sd0           sd1           sd3          cpu
 tin tout Kps tps serv  Kps tps serv  Kps tps serv  Kps tps serv  us sy wt id
   0    1   0   0    0    3   0   50    2   0   51    2   0   63  18  7  2 73
   0   16   0   0    0   23   4   43    0   0    0    0   0    0  19  9 13 60
   0   15   0   0    0   99  16   38    0   0    0    0   0    0  32 17 49  2
   0   16   0   0    0   93  15   37    0   0    0    0   0    0  43 17 40  0
   0   16   0   0    0  111  17   40    0   0    0    0   0    0  41 17 41  1
   0   16   0   0    0  117  17   36    0   0    0    0   0    0  40 22 37  1
   0   16   0   0    0  103  18   50    0   0    0    0   0    0  29 18 51  2
   0   16   0   0    0    6   1   85    0   0    0    8   1  116  20  7  0 73
```

The new variant "iostat -x" provides extended statistics, and is easier to read when a large number of disks are being reported since each disk is summarized on a separate line. The values reported are the number of transfers and KB per second, with read and write shown separately; the average number of commands waiting in the queue; the average number of commands actively being processed by the drive[1]; the service time described above; and the percentage of the time that there were commands waiting in the queue, and commands active on the drive.

```
% iostat -txc 5
                              extended disk statistics        tty         cpu
disk       r/s  w/s   Kr/s   Kw/s wait actv  svc_t  %w  %b  tin tout us sy wt id
fd0        0.0  0.0    0.0    0.0  0.0  0.0    0.0   0   0    0   77 42  9  9 39
sd0        0.0  3.5    0.0   21.2  0.0  0.1   41.6   0  14
sd1        0.0  0.0    0.0    0.0  0.0  0.0    0.0   0   0
sd3        0.0  0.0    0.0    0.0  0.0  0.0    0.0   0   0
                              extended disk statistics        tty         cpu
disk       r/s  w/s   Kr/s   Kw/s wait actv  svc_t  %w  %b  tin tout us sy wt id
fd0        0.0  0.0    0.0    0.0  0.0  0.0    0.0   0   0    0   82 38 17 44  2
sd0        0.0 14.8    0.0   90.9  0.0  0.6   38.6   0  56
sd1        0.0  0.0    0.0    0.0  0.0  0.0    0.0   0   0
sd3        0.0  0.0    0.0    0.0  0.0  0.0    0.0   0   0
                              extended disk statistics        tty         cpu
disk       r/s  w/s   Kr/s   Kw/s wait actv  svc_t  %w  %b  tin tout us sy wt id
fd0        0.0  0.0    0.0    0.0  0.0  0.0    0.0   0   0    0   84 37 17 45  1
sd0        0.0 16.8    0.0  102.4  0.0  0.7   43.1   2  61
sd1        0.0  0.0    0.0    0.0  0.0  0.0    0.0   0   0
sd3        0.0  1.0    0.0    8.0  0.0  0.1  114.3   2   4
```

How To Decide That A Disk Is Overloaded

The key value to watch is the service time (svc_t). This is the time taken to service an I/O request to this drive, including time spent waiting in the queue because other requests were being processed. At very light loads spurious large service times are seen, so a threshold load level is also important. From various sources, particularly the NFS server LADDIS benchmark, it is generally recognized that I/O service times of more than 50ms are considered too slow. Disks that are over 30% busy averaged over a 30 second period should have their service times checked. A script that can be used to monitor either the local machine or a remote server is shown in Figure 2 on page 105. It runs iostat -x 30 on the machine being monitored, and a nawk script on the local

1. This will be less than 1.0 unless the drive uses the IPI controller or is a SCSI drive with tagged command queueing support.

machine. When started it goes into the background and when it sees a busy disk it prints a warning message to the console and plays an audio file to attract the users attention (by default the sound indicates that “performance just went down the toilet”).

Figure 2 iomon - disk monitoring script for Solaris 2

```
#!/bin/csh
# Adrian.Cockcroft@corp.sun.com 4th Nov 1993
# requires Solaris 2 specific version of iostat
# warn the operator that a disk is overloaded
# cmdline argument is optional server name for remote monitoring
#
# hardwired arguments:
# 30 second monitoring interval for iostat
# monitor disks that contain sd in the name to avoid floppies
# look for disks over 30% busy with more than 50.0ms service time
# play an appropriate sound to indicate performance just went down
# the toilet, /usr/demo/SOUND/sounds/flush.au needs SUNWaudmo pkg
# print out message to the local system console
#
if ($1 == "") then
    set CMD="iostat -x 30"
    set HOST=`hostname`
else
    set CMD="rsh $1 exec iostat -x 30"
    set HOST=$1
endif
#                                    disk    %b          svc_t
exec $CMD | nawk -v host=$HOST '$1 ~ "sd" {if ($10 > 30 && $8 \
> 50.0) { printf("%s: ", host); print; system("audioplay \
/usr/demo/SOUND/sounds/flush.au"); }}' >/dev/console &
```

The output doesn’t include the iostat header line, but it is easy to get a reminder if it is needed. Here is an example of starting and testing iomon.

```
% iomon
[2] 18220 18221
% mkfile 50M /var/tmp/JUNK; rm /var/tmp/JUNK
hostname: sd3      1.9 25.8    9.4 1195.3  4.8  2.9  280.8  95 100
% iostat -x 1 1
                                  extended disk statistics
disk      r/s  w/s   Kr/s   Kw/s wait actv  svc_t  %w  %b
...
```

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. Tests have shown that with the 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 can support about three to four heavily loaded disks per bus.

For sequential accesses there is an obvious limit on the aggregate bandwidth that can be sustained by a number of disks. For random accesses the bandwidth of the SCSI bus is much greater than the aggregate bandwidth of the transfers, but queueing effects increase the latency of each access as more disks are added to each SCSI bus. Database applications need low latencies so for best performance three or four active disks per SCSI bus is the recommended maximum.

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 1.0 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.

Note – Online: DiskSuite 2.0 runs on Solaris 2 and does not need this fix.

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.

1. See "The Disk Tuning Performed For SunOS 4.1.1" on page 95, and the tunefs manual page.

- 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 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

9

This chapter looks at some SPARC implementations to see how the differences affect performance. A lot of hard-to-find details are documented. See also the recent book "Multiprocessor System Architectures, Ben Catazaro, SunSoft Press".

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 specifies this interface and it is controlled by the independent SPARC International organization. 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)

The ISA is defined by the SPARC Architecture Manual, SPARC International has published Version 7, Version 8 and Version 9, the IEEE have a draft standard based on Version 8, IEEE P1754. Version 9 defines major extensions including 64 bit addressing in an upwards compatible manner for user mode programs. Version 9 based systems are expect to appear during 1994 and 1995.

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 parameterized 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 17 on page 113.

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

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

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”

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 SunPro SPARC compiler optimizer performs tail recursion elimination and leaf routine optimization 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 other systems with write back caches will not stall (unless a cache line needs to be updated).

The SPARC V9 architecture supports a new multiple level trap architecture. This greatly reduces the administrative overhead of register window traps, since the main trap handler no longer has to check for page faults. This is expected to increase the relative performance boost of register windows by reducing the trap time.

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 vectorizable 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 coroutine or threads library. When running SunOS a context switch rate of 1000 per 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 kernel 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.

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.

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 specialized instructions. The time taken to complete each floating point operation in cycles is shown in Table 18 on page 117.

Table 17 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 model 40 (sun4m)	40MHz	SuperSPARC	SuperSPARC
SPARCsystem 10 & 600 model 41 (sun4m)	40MHz	SuperSPARC	SuperSPARC
SPARCsystem 10 & 600 model 51 (sun4m)	50MHz	SuperSPARC	SuperSPARC
SPARCclassic & SPARCstation LX (sun4m)	50MHz	MicroSPARC	MicroSPARC
SPARCcenter 1000 & 2000 (sun4d)	50MHz	SuperSPARC	SuperSPARC
Cray S-MP & FPS 500EA (sun4)	66MHz	BIT B5000	BIT B5010
MicroSPARC II	70+MHz		
SPARC PowerUP (sun4c)	80MHz	Weitek	Weitek on-chip

Sun engineering teams designed many of 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 17 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.
- A new MicroSPARC II design with higher performance and even higher integration than MicroSPARC has just been announced by Fujitsu.

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. The Ross division of Cypress (which produced all the SPARC parts) was recently bought by Fujitsu.

The Weitek SPARC PowerUP Upgrade

Weitek have taken the basic design used in the MB86903 and modified it to double the internal clock rate and provide 24KB of on-chip caches. This is sold as a chip level upgrade for SPARCstation 2 and IPX machines, and its 80MHz operation runs some programs between 1.5 and 2 times faster than the basic 40MHz SPARCstation 2 or IPX.

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 18. 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.

1. See "The SuperSPARC Microprocessor Technical White Paper"

SuperSPARC is shipping at 33, 36, 40 and 50MHz, and has been announced at 60MHz by Texas Instruments. A derived SuperSPARC II design is part of the SPARC futures roadmap for higher performance.

Cypress/Fujitsu Ross 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. HyperSPARC has just started shipping at 55MHz and is targeting 66MHz and 80MHz. The performance of a 66MHz HyperSPARC is comparable to a 50MHz SuperSPARC. Fujitsu recently bought Ross from Cypress.

UltraSPARC

Sun recently announced a SPARC roadmap for the next five years. Apart from higher clock rates the next major development will be the superscalar UltraSPARC, which implements the 64bit SPARC V9 architecture and will run at well over 100MHz.

Low Cost Implementations

The main low cost SPARC chips are the Fujitsu SPARClite, a 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.

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 (optimized for the SPARCstation 2) or -cg92 (optimized 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 iterative FPU that is data dependent and minimum, typical and maximum times are given.

Table 18 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 19 Number of Register Windows and Integer Cycles per Instruction

Instruction	Fujitsu/LSI #1, #2	Cypress/ Fujitsu #3	MicroSPARC (&B5000)	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 (1)	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	1
std (64 bit double)	4	4	3 (2)	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

10

Multiprocessor machines introduce yet another dimension to performance tuning. Sun has been shipping multiprocessor servers for some time but the introduction of Solaris 2 and the desktop multiprocessor SPARCstation 10 is bringing 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 utilization 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?

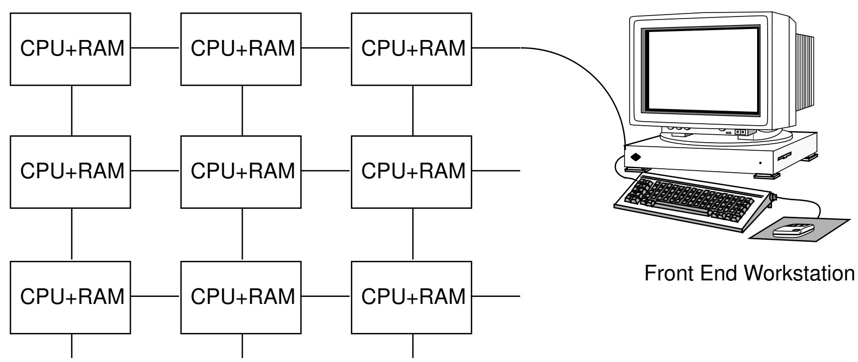
At any point in time there are CPU designs that represent the best performance that can be obtained with current technology at a reasonable price. The cost and technical difficulty of pushing the technology further means that the most cost effective way of increasing computer power is to use several 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 hard 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

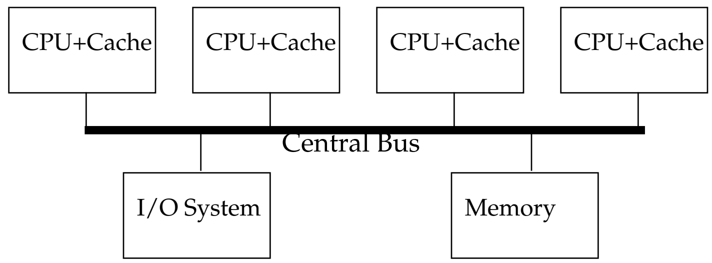


These can be thought of as a network of uniprocessors packaged into a box. Each processor has its own memory and data must be explicitly copied over the network to another processor before it can be used. The benefit of this is that there is no contention for memory bandwidth to limit the number of processors 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; and Meiko CS2 which has SuperSPARC processors running Solaris 2 on each node; the Intel iPSC Hypercube which has Intel 386, 486 or 860 processors; a myriad of other Transputer based machines and 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), the 8 processor SPARCserver 1000 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 synchronization 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 characterize 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 and can be tuned to scale well with large numbers of processors. The same

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

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. As Solaris 2 is tuned for more concurrency some of the remaining code locks are turned into data locks, and "large" locks are broken down into finder grained locks. Tools exist to monitor the lock contention in the kernel, but access to kernel source code is required to make sense of the results. If mutex access routines appear right at the top of the list of a kernel profile then contention may be occurring[1]. There is a trade-off between having lots of mutexes for good MP scalability, and few mutexes for reduced CPU overhead on uniprocessors.

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

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

1. See "Kernel Profiling Using Kgmon In Solaris 2" on page 74.

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 optimized for a standard transaction size of 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 optimized 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 in 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 inter-board backplane. The backplane bus is called XDbus. On the SPARCserver 1000 there is a single XDBus and on the SPARCcenter 2000 there is a twin XDbus (one, two or four can be implemented with different board sizes using the same chipset).

Table 20 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
Single XDBus	320Mbytes/s	250Mbytes/s	250Mbytes/s
Dual XDbus	640Mbytes/s	500Mbytes/s	500Mbytes/s
Quad XDbus	1280Mbytes/s	1000Mbytes/s	1000Mbytes/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.

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 organized 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 organization so the intermediate buffer is not needed. 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.

3. This is known as *reflective memory*.

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.

The system configuration can be seen on SPARCserver1000 and SPARCcenter2000 machines with the /usr/kvm/prtdiag command.

Measuring And Tuning A Multiprocessor With 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 21 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

Measuring And Tuning A Multiprocessor With Solaris 2

Solaris 2.0 only runs on uniprocessor machines (Sun4c kernels). Solaris 2.1 was tested and tuned for machines with up to four CPUs and Solaris 2.2 was tested and tuned for up to 8 CPUs. Solaris 2.3 will support the full SPARCcenter 2000 configuration after extended testing, shortly after the regular Solaris 2.3 ships.

CPU Control Commands - psrinfo and psradm

Some new commands were implemented in Solaris 2.2. Psrinfo tells you which CPUs are in use, and when they were last enabled or disabled. Psradm actually controls the CPUs. Note that clock interrupts always go to CPU 0[1], even if it is disabled. Solaris 2.3 reintroduces a more useful version of mpstat.

Cache Affinity Algorithms

When a system that has multiple caches is in use a process may run on a CPU and load part of itself into that cache then 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

1. On sun4d machines the system CPU defaults to CPU 0 but can be configured. See which board has the console connected to it.

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 has been developed. In the case of the SPARCcenter 2000 this algorithm is managing up to 20Mbytes of cache and has a significant effect on performance. The algorithm works by moving jobs from the central run queue to a private run queue for each CPU. The job stays on a CPU's run queue unless another CPU becomes idle, looks at all the queues, finds a job that hasn't run for a while on another queue and migrates the job to its own run queue.

Programming A Multiprocessor

Mp Programming With SunOS 4.1.2 And 4.1.3

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

MP Programming With Solaris 2.0 And 2.1

The programmers libraries are not present in these releases.

MP Programming With Solaris 2.2

A multithreaded programmer interface is included in this release, and several system libraries have been made reentrant, or MT-safe.

MP programming With Solaris 2.3

Unix International (SVR4 ES/MP) and a POSIX standards committee (1003.4a) are defining API's for multithreaded programming. Solaris 2.3 includes a conforming implementation of the POSIX threads standard and has many more MT-safe system libraries.

SunPro has made available an early access release of their multithreaded debugger and automatic parallelizing Fortran compiler to some customers[1].

1. See the "SPARCserver and SPARCcenter Performance Brief" for some parallelized benchmark results.

Deciding How Many CPU's To Configure

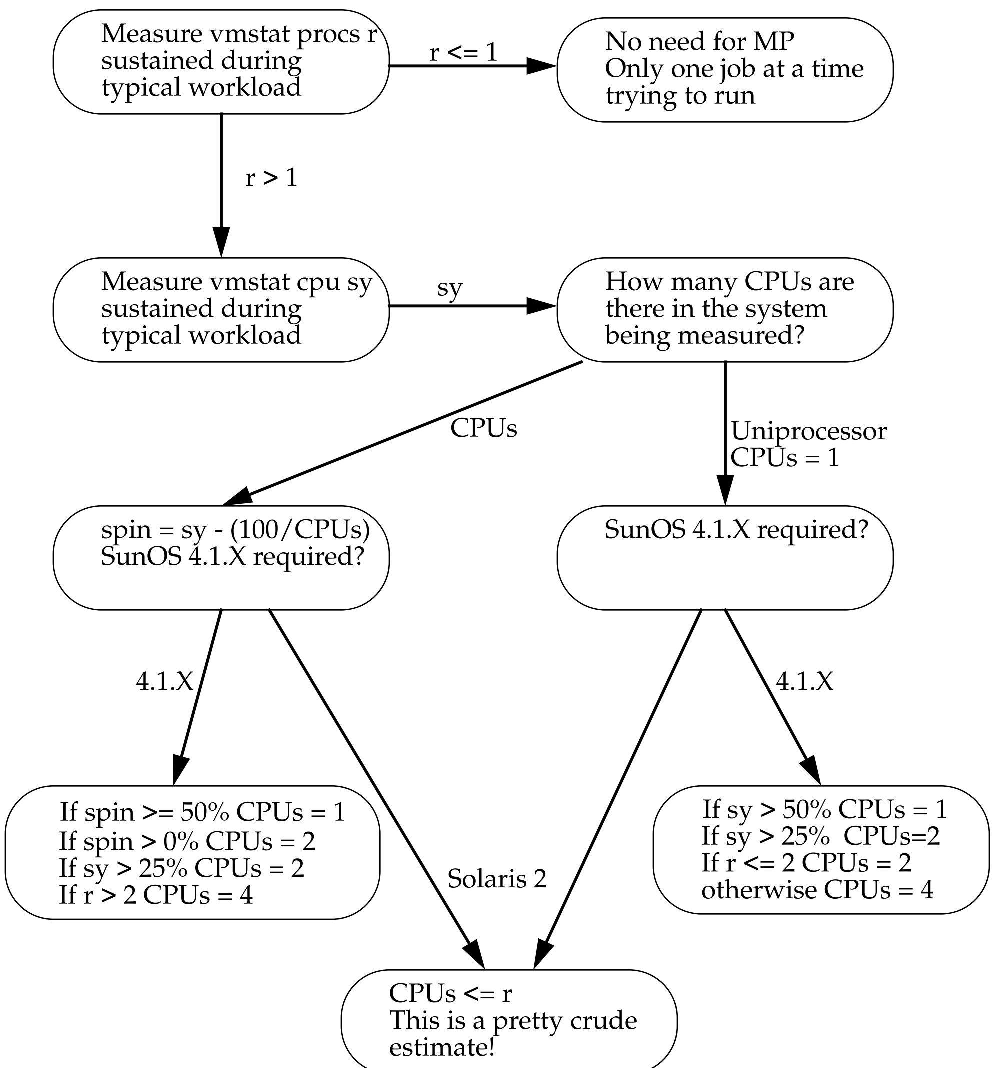
If you have an existing machine running a representative workload then you can look at the vmstat output to come up with an approximate guess for the number of CPU's that could be usefully employed by the workload. Interpreting vmstat depends upon the number of CPU's configured in the existing machine 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 below to guide you through this process. The results should be used as input into a decision and should not be taken too seriously. The end result is an estimate of the amount of process level concurrency in the workload.

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.

Note – If you use a Solaris 2 measured workload with the flowchart below then you need to add the number of CPUs in the system to `r` if `r` is nonzero, or work out `r` from the CPU times if it is zero.

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
```

Interrupt Distribution Tuning

On the SPARCserver 1000 and SPARCcenter 2000 there are up to 10 independent SBuses, and there is hardware support for steering the interrupts from each SBus to a specific processor.

The algorithm used in Solaris 2.2 is that the clock interrupt is permanently assigned to CPU 0, to obtain good cache hit rates in a single cache. The clock presents a relatively light and fixed load at 100Hz so this does not significantly unbalance the system. To balance the load across all the other CPUs a round robin system is used, whereby all interrupts are directed to one CPU at a time. When it takes the first interrupt, it sends a special broadcast command over the XDBus to all the SBus controllers to direct the next interrupt to the next CPU. This balances out the load but when there is a heavy interrupt load from a particular device it is less efficient from the point of view of cache hit rate.

The algorithm can be switched to a static interrupt distribution, whereby each SBus device is assigned to a different CPU. For some I/O intensive workloads this has given better performance, and it is the default in Solaris 2.3.

The kernel variable that controls this is called do_robin, and it defaults to 1 in the sun4d kernel architecture of Solaris 2.2 If it is set to 0 in /etc/system then the static interrupt distribution algorithm is used.

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. This board is not needed on Solaris 2, since the NFS code is multithreaded, and better NFS server performance is obtained by using a multiprocessor SPARC machine.

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. The SPARCstation figures are in the SPARCstation 2 Performance Brief.

Table 22 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

FDDI Interfaces

There are two FDDI interfaces that have been produced by Sun, and several third party VMEbus and SBus options as well. FDDI runs at 100Mbits/s so has ten times the bandwidth of ethernet. When running an NFS client over FDDI to a server it is a good idea to increase the number of “biod” daemons in SunOS 4.X and to increase the number of NFS asynchronous kernel threads in Solaris 2 from the default of 4 to 16. This allows more outstanding requests to be sent to the server, and provides better throughput.

VMEbus FDDI/DX

This is the original Sun FDDI board, and was one of the first available in the industry in 1989. It is bottlenecked by both the VMEbus interface and the on-board 68020 that runs the station management protocol. A peak throughput of about 30Mbits/s can be expected. The FDDI/DX has a class A dual attach interface, which connects directly into the FDDI rings.

SBus FDDI/S

This is the current Sun FDDI board. On low end machines it is limited by the host CPU performance, but on SuperSPARC based machines a full 100Mbits/s is possible under certain conditions. The device driver for the card can be tuned to improve performance by changing buffer allocations.

FDDI performance in Solaris 2.2 is not well tuned with the default parameters, but substantial improvements have been made for 2.3. Sun has published results for the SPEC NFS benchmark "LADDIS" using FDDI and an alpha release of Solaris 2.3, including the following settings in /etc/system, which are applicable to Solaris 2.2 as well.

For the NFS clients the number of threads defaults to eight, which works well with ethernet, but for FDDI the optimal number is 16.

set nfs:nfs_asynch_threads = 16

For the FDDI device itself set the number of receive buffers as shown below.

set bf:bf_nrmds1 = 32 (max 256)

set bf:bf_nbufs = 64 (must be > bf_nrmds1)

Do a netstat -k and look under bf0. If rx_notavail count is high, set bf_nrmds1 high. Adjust bf_nbufs as well (see restrictions under bf_nbufs above). If tx_notavail is high, set bf_nbufs high.

Note – The NFS protocol itself limits throughput to about 3MB/s because it has limited pre-fetch and small block sizes. The upcoming NFS version 3 protocol allows larger block sizes and other changes to improve performance on high speed networks. Lower level protocols like ftp can use larger TCP/IP transmit and receive windows to run at full speed over FDDI.

Using NFS Effectively

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

This last document is part of the Solaris 2.3 SMCC hardware specific manual set. It contains a good overview of how to size an NFS server configuration, but its tuning recommendations should be considered a first draft, as there are many errors and inconsistencies. It actually covers SunOS 4.1.3/Solaris 1.1 NFS server tuning as well as Solaris 2.

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 about 30 `nfsds` should be configured *per ethernet* on both SunOS 4 and Solaris 2. 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 as long as you don’t run out of process table entries. Some high end LADDIS results use over 300 nfsds.

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 2.2

This is the basic reference on Solaris 2 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.

The Art of Computer Systems Performance Analysis, by Raj Jain

This recent book is a comprehensive and very readable reference work covering techniques for experimental design, measurement, simulation and modelling. Published by Wiley, ISBN 0-471-50336-3.

Building and Debugging SunOS Kernels, by Hal Stern

This Sun white paper provides an in-depth explanation of how kernel configuration works in SunOS 4.X, with some performance tips.

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 used in the SPARCserver 490 and subsequent server designs.

Computer Architecture - A Quantitative Approach, Hennessy and Patterson

The definitive reference book on modern computer architecture.

The Design And Implementation Of The 4.3BSD UNIX Operating System, Leffler, McKusick, Karels and Quarterman

This book describes the internal design and algorithms used in the kernel of a very closely related operating system. SunOS 3.X was almost a pure BSD4.3, SunOS 4 redesigned the virtual memory system, and UNIX System V Release 4 is about 80% based on SunOS 4 with about 20% UNIX System V.3. Solaris 2 has been further rewritten to support multiple processors. Despite the modifications over time this is a definitive work, and there are few other sources for insight into the kernel algorithms. ISBN 0-201-06196-1.

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.

High Performance Computing, Keith Dowd

This is a very recent book that covers the architecture of current high performance workstations, compute servers and parallel machines. It is full of examples of the coding techniques required to get the best performance from the new architectures, and contrasts vector machines with the latest microprocessor based technologies. The importance of coding with caches in mind is emphasized. The book is essential reading for Fortran programmers trying to make numerical codes run faster, and is highly recommended for all programmers. Published by O'Reilly ISBN1-56592-032-5.

Managing NFS and NIS, Hal Stern

Network administration and tuning is covered in depth. An essential reference published by O'Reilly ISBN0-937175-75-7.

Multiprocessor System Architectures, Ben Catazaro, SunSoft Press

This book is a great reference to all the details of SPARC based system hardware. It describes all the chip sets and system board designs, including the latest multiprocessor systems.

Networks and File Servers: A Performance Tuning Guide

This is the best of the network tuning white papers. It is starting to get a little out of date

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.

Optimisation in the SPARCompilers, Steven Muchnick

The definitive description of the SPARC compilers and how they work. There is a Sun white paper and several conference proceedings on this subject.

Performance Tuning an Application

Part of the SPARCworks online answerbook with the compiler products and the Teamware manuals. Defined are new tools that may be unfamiliar to many users doing software development.

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 standard IEEE 1496. An SBus developers kit is available from Sun, that contains the SBus specification, example drivers, lots of related documentation a Forth tokenizer program and some sample mechanical parts.

A Scalable Server Architecture for Department to Enterprise - The SPARCserver 1000 and the SPARCcenter 2000

The architecture white paper for the SS1000 and SC2000. The paper progresses logically into greater and greater amounts of detail, so start at the beginning and read until you've had enough. May 1993.

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 when 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.

SMCC NFS Server Configuration and Performance Tuning Guide

This is part of the Solaris 2.3 SMCC hardware specific manual set. It contains a good overview of how to size an NFS server configuration, but its tuning recommendations should be considered a first draft, as there are many errors and inconsistencies. It actually covers SunOS 4.1.3/Solaris 1.1 NFS server tuning as well as Solaris 2.

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 and has been superceded by the SunOS 5.2 Guide to Multithread Programming which is part of the Solaris 2.2 manual set.

Solaris XIL 1.0 Imaging Library White Paper

Describes the XIL Imaging Library functions and how specific applications may require XIL functions. Includes glossary of imaging and video terminology. Feb. 1993.

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 has been published by SPARC International and includes upwards compatible 64bit extensions and a revised kernel level interface.

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’s SPARC Technology Business Unit. 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!

SPARCclassic X Performance Brief

Xmarks, X11perf v1.2, and Xbench numbers for the SPARCclassic X Terminal. Descriptions of the benchmarks and product highlights are included. July 1993.

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 1000 Compute White Paper

This paper covers the requirements of a compute server, a hardware overview, Solaris features, and developing applications for multi-threading and multi-processing. May 1993.

SPARCserver 1000 Performance Brief

SPECint92, SPECfp92, SPECrate_int92, SPECrate_fp92, Linpack1000, AIM3, 097.LADDIS (SFS), and TPC-A information for the SS1000. Solaris 2.2, May 1993.

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 parallelized SPEC92, SPECthroughput, AIM III and parallelized Linpack benchmarks using the Solaris 2.1 operating system.

SPARCserver Sizing Guide for X terminals

This guide answers the question, “How many SPARCclassic X based Frame or Wabi users should be configured on a SPARCserver 10 or 1000 with 2 or 4 CPU’s and from 64 to 256MB of RAM”. Aug 1993.

SPARCstation 2GS / SPARCstation 2GT Technical White Paper

This describes two of 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. January 1991.

SPARCstation 10 Performance Brief

Contains SPECint92, SPECfp92, Dhrystone V1.1, Linpack DP 1000, SPECrate_int92, SPECrate_fp92 benchmark information for the SS10 30LC,40, 41, 51, 402, 512, and 54. Solaris 2.2, April 1993.

SPARCstation 10 Product Line Technical White Paper

This paper explains the differences between various uniprocessor and multiprocessor SPARCstation 10 models and provides results of many application and benchmark tests on each model. It also contains a simplified architectural overview of the machine. June 1993.

SPARCstation 10 System Architecture - White Paper

A comprehensive overview of the SPARCstation 10.

SPARCstation 10SX Graphics Technology, A White Paper

This is a very hot product and the white paper does it justice. It does a nice job of discussing why the SX is a good fit for various markets, with the right number of buzz-words and right amount of detail. It then goes into the architecture and software implications. July 1993.

SPARCstation 10ZX and SPARCstation ZX Graphics Technology, A White Paper

The paper is broken down into features, architecture, and SW. The paper is chock full of clearly presented information. You do need to be a graphics person to fully appreciate it. July 1993.

SPARCstation LX and SPARCclassic Performance Brief

This is the complete SPARCstation LX and SPARCclassic Performance Brief, it is based upon Solaris 2.1 and is dated November 1992. There have been significant performance improvements in some areas since then.

SPARCstation ZX, SPARCstation 10ZX and SPARCstation 10 TurboGXplus Graphics Performance Brief

GPC, X11perf, and primitive test benchmarks for the products listed in the title. Descriptions of the benchmarks and product highlights are included. July 1993.

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 by Sun.

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.

TurboGXplus Graphics Technology, A White Paper

Lots of good information, including features, architecture, and software. Clearly written, a good solid description of the TGX. July 1993.

Virtual Swap Space in SunOS by Howard Chartock, Peter Snyder

The concept of swap space in SunOS 5 has been extended by the abstraction of a virtual swap file system, which allows the system to treat main memory as if it were backing store. Further, this abstraction allows for more intelligent choices to be made about swap space allocation on the current system. This paper contrasts the existing mechanisms and their limitations with the modifications made to implement virtual swap space. May 1992.

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 and later in a modified form as a performance tuning chapter.

Revision History

Part Number	Revision	Date	Comments
	-01	July 1991	First complete draft for review
	-02	August 1991	Cache details corrected
	-03	September 1991	More corrections, presented at Sun User '91, distributed
	-04	September 1992	Update to 4.1.3, S600MP and SS10, draft issued internally
801-4872-01	-05	December 1992	Update to Solaris 2.1, SC2000, LX and Classic, distributed
801-4872-01	-06	August 1993	Update to include Solaris 2.2, SS1000, SS10 models. Major update, draft issued internally.
801-4872-07	A	October 1993	Update to include Solaris 2.2, 2.3, SS1000, SS10 models. Minor corrections, reissued as offcial SMCC White Paper
801-4872-07	B	December 1993	Table widths corrected, minor updates and fixes.

A part number was assigned to the -05 revision after it was distributed externally. A new front page was added to the -05 document for Sun internal duplication and distribution. The part number and revision was not set up correctly until the -07 revision A release, which accounts for the inconsistencies shown above.

Sun Microsystems
Computer Corporation

A Sun Microsystems, Inc. Business

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

Australia: (02) 413 2666
Belgium: +32 2 759 38 11
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) 3221-7021
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: (01) 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

Printed in USA

10/93