Sun Performance Tuning Overview

sun
microsystems

Written by

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

Phone: +44 223 420421
US Email: adrian.cockcroft@uk.sun.com
UK Email: adrian.cockcroft@sun.co.uk

For

Sun User '91, 10-12 September 1991, NEC, Birmingham, UK

Version

1.0	25 July 91	First complete draft for review
1.1	5 August 91	Cache details corrected
1.2	5 September 91	More corrections, Sun User '91

Introduction

1

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.

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 rest of this paper looks at each layer in turn and the appendix has details of the references.

Table 1-1 System Broken Down Into Tuning Layers

Layers	**Variables**
Sourcecode	Algorithm, Language, Programming model, Compiler
Executable	Environment, Filesystem
Database	Buffer sizes, lock time-outs
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
Network	Protocol, hardware, tuning

Sourcecode

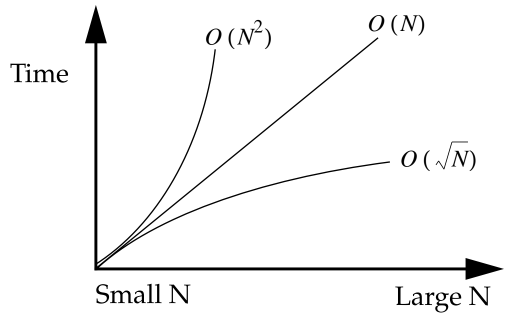
2

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

Algorithms

Of course all programmers keep a good book of algorithms and always use the most efficient one for the job don't they? In practise many algorithms are made up on the fly or are as simple to code as possible. More efficient algorithms are often much more complex to implement and may need to be retrofitted to a system that is being tuned. There are some general classes of algorithms with behaviors as shown below.

Figure 2-1 Performance vs. problem size for different classes of algorithms



The notation O(N) means that as N increases the time taken increases linearly. Higher order increases such as increasing by N squared or exponential increases need 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 dataset 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.

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 is OK for small drawings and it is OK 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 wish 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 (more on that later) 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.

Programming Model

There is often a conceptual framework underlying a system 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 by a customer. 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. 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 are sometimes so tied up in contemplating their own navel that brute force increases in CPU and I/O throughput are the only thing that has a significant effect.

The moral is that if you come across a system 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.

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

Fortran

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

Assembler

In theory assembler is the fastest language possible. In practise 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.

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!

C++ should be used whenever an object oriented design is called for. C could be used in most cases but the end result is that sloppy coding and shortcuts combined with the extra level of complexity make the resulting systems very hard to debug and hence give optimisers a hard time. Anyone who has had to use a C debugger to debug a C++ program (i.e. the output from the C++ preprocessor) will know that the code gets pretty hard to follow.

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 SPARCworks compilers from Sun tend to have the largest user base and the best optimisation and robustness on large programs. There are an increasing number of 3rd party alternatives as well. Using compilers effectively to tune code is not covered in this paper since it has been addressed in great depth in previous publications, listed below.

- Numerical Computation Guide, supplied with SunC and SunFortran
- You and Your Compiler, by Keith Bierman
- Several papers by Steven Muchnick, including one at Sun User 91.
- SPARCompiler Optimisation Technology 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.

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

Mapped files

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

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. See the following references.

- SunOS 4.1 Performance Tuning
- mmap and aioread, aiowrite manual pages

Linking, localisation of reference and libraries

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

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

```
% f77 -fast -o fred fred.f -lc -Bstatic
```

This explicitly links in the `libc` library then makes everything else static. The order of the arguments is important. This is one of those tricks that is hard to work out and not documented anywhere else so make a note of it!

Tuning Shared Libraries

When using static linking the library is accessed as an archive of separate object files and only the files needed to resolve references in the code are linked in. This means that the position of each object module in memory is hard to control or predict. For dynamic linking the entire library is provided at run time regardless of which routines are needed. In fact, the library is demand paged into memory as it is needed. Since the object modules making up the library are always in the same order the library can be reordered 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.

Executable

3

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

Customising the execution environment

Limits

The execution environment is largely controlled by the shell. There is a `limit` command which can be used to constrain a program that is hogging too many resources. The default SunOS 4.1.1 resource limits are shown in table 3-1.

Table 3-1 Resource Limits

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

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

```
% limit coredumpsize 0
```

To run programs that use vast amounts of stack space:

```
% limit stacksize unlimited
```

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

```
% limit descriptors 256
```

Tracing

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

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. Some of the commentary is a little speculative!

```
% 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                                         check ld.so is OK?
mmap (0, 40960, 0x5, 0x80000002, 3, 0) = 0xf77e0000             map in ld.so code to memory?
mmap (0xf77e8000, 8192, 0x7, 0x80000012, 3, 32768) = 0xf77e8000 map in ld.so 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 () = 1434                                                get group id
open (“/etc/ld.so.cache”, 0, 05000000021) = 3                   open the shared library cache
fstat (3, 0xf7fff750) = 0
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
fstat (3, 0xf7fff750) = 0                                       /usr/openwin/lib so look
mmap (0xf7802000, 8192, 0x3, 0x80000012, 4, 0) = 0xf7802000     there first
getdents (3, 0xf78000d8, 8192) = 1488                           Get some directory entries
getdents (3, 0xf78000d8, 8192) = 0
```

```
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
close (1) = 0                                                         close stdout
close (2) = 0                                                         close stderr
exit (0) = ?                                                          exit program
%
```

Timing

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

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

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

The effects of underlying filesystem type

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

UFS

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

tmpfs

This is a real RAMdisk 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 use is to un-comment the line in /etc/rc.local for `mount /tmp`. Some operations, such as file locking, are not supported in a tmpfs filesystem so applications that lock files in /tmp will misbehave.

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

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

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

NFS

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

References

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

Databases and Configurable Systems 4

This layer 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.

Get Hold Of 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.

Get DBE

Sun has a version of SunOS tuned to run on systems with large amounts of memory and databases. It is called DBE - Database Excelerator and there are two versions. DBE 1.0 for SunOS 4.1, and DBE 1.1 for SunOS 4.1.1. It is cheap at a few hundred pounds for media, manual and site licence, and it 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 and 16 Mb is the minimum configuration supported.

Basic Tuning Ideas

I keep coming across untuned copies of Oracle so some basic recommendations on the first things to try may be useful. They probably 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 200Kbytes but it can be increased to be bigger than the entire dataset if needed. I would increase it to about 10% of the total RAM in the machine as a first try.

Using raw disk rather than filesystems

You should reserve some 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 give oracle the raw devices rather than files in the usual filesystem as the standard data, log1 and log2 files. 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.

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

Kernel

5

This layer is concerned with variables that can be changed by a system administrator building or tuning a SunOS kernel. The scope for tuning SVR4 kernels is greatly reduced since many operations have been automated and there is no need to recompile the kernel.

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.

Table 5-1 Tuning Older SunOS Releases

Release	Extra Tuning Required (apart from maxusers)
SunOS 4.0	Number of streams, number of mbufs
SunOS 4.0.1	Same as SunOS 4.0
SunOS 4.0.3	Add PMEGS patch tape, set handspread
SunOS 4.1	Add PMEGS patch tape or DBE-1.0, set handspread
SunOS 4.1.1	Add DBE-1.1, increase buffer cache

Maxusers

The maxusers parameter is set in the kernel configuration file. Many parameters and kernel tables are derived from it. It is intended to be derived from the number of users on the system but this usually results in too small a value. It defaults to 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 around 200 and a very rough guideline would be to set it at anything up to twice the number of Megabytes of RAM in the system.

Directory Name Lookup Cache (dnlc)

This is sized using Maxusers and a large cache size significantly help NFS servers which have lots of clients.

Pstat

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

```
% pstat -T
217/582 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.

Inode Cache

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

Buffer Cache

The buffer cache is used to cache all disk I/O in SunOS 3 and BSD Unix. In SunOS 4 and SVR4 it is used to cache inode and cylinder group related disk I/O only. It is set via the kernel variable `nbuf` and is usually set to around 32. It should be increased to 64 for machines with up to four disks and 112 for machines with more than four disks. Details on tuning the buffer cache are provided in the following reference.

- Tuning the SPARCserver 490 for Optimal NFS Performance

Paging, MMU algorithms and PMEGS

Vmstat

The paging activity on a Sun can be monitored using vmstat.

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

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

Memory

Report on usage of real memory. Avm is used in SunOS 3 and other Berkeley based Unix implementations but has no meaning in SunOS 4 and SVR4. This is not what it says in the manual page!

avm	active virtual memory is a historical measure that is always set to zero.
fre	size of the free list in Kbytes

Page

Report information about page faults and paging activity. The information on each of the following activities is averaged each five seconds, and given in units per second.

re	page reclaims.
at	number of attaches.
pi	kilobytes per second paged in
po	kilobytes per second paged out
fr	kilobytes freed per second
de	anticipated short term memory shortfall in Kbytes
sr	pages scanned by clock algorithm, per-second

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

PMEGs

Current Sun machines use a hardware memory management unit which acts as a large cache for memory translations. It is much larger than the MMU translation cache found on other 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 64 physical 4Kb page translations. A SPARCstation 1 has a cache containing 128 PMEGS so a total of 32Mb of virtual memory can be cached. The number of PMEGS on each type of Sun is shown in Table 5-2.

Table 5-2 MMU Characteristics

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

In SunOS 4.0.3 and SunOS 4.1 there were two independent problems which only came to light when Sun started to ship 4Mbit DRAM parts and the maximum memory on a SPARCstation 1 went up from 16 Mb to 40Mb. Memory prices also dropped and many more people loaded up their SPARCstation 1,1+, SLC or IPC with extra RAM. When the problem was discovered a patch was made for SunOS 4.0.3 and SunOS 4.1. The patch is incorporated into DBE 1.0, DBE 1.1 and SunOS 4.1.1 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

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 hotline to get hold of the PMEGS patch.

Paging

The scanning algorithm which tries to find pages to put onto the free list uses a parameter called `handspread` that should be set to approximately a quarter of the RAM available in the machine. Unfortunately, in the old days of Berkeley Unix, when a 2 Mb VAX 750 was a big machine, the code that set up handspread clamped it to a maximum of 2 Mb. This code persisted until SunOS 4.1.1 so handspread needs to be patched on any machine that has much more than 8 Mb of RAM and is running SunOS 4.1 or before.

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

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

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

The effect of the problem is that pages that are actively being used are more likely to be put on the free list if handspread is too small, causing extra work for the kernel. If the system has a lot of memory installed an stop/go effect can occur where the scanning algorithm will be triggered by a shortage of memory and it will free every page it finds until there is masses of free memory then it will go to sleep again. On a properly configured system it should be trickling around continuously at a slow rate tidying up the pages that haven't been referenced for a while.

The 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 256K `lotsfree`) the hands start to move round the clock at a slow rate which increases to 100 pages per second (`slowscan`), as the free list gets down to 100K (`desfree`). The first hand resets a flag in the page which is set every time the page is accessed and the second hand, when it gets round to the page checks to see if the page has been accessed since the first hand got there. If not then it is moved to the free list and if the page contained modified data the page is written to disk. If there is a lot of memory in use then the chances of an individual page being accessed in a given time span are reduced so the spread of the hands must be increased to compensate. If the shortage of memory gets worse (less than 100K `desfree`) then the scan rate increases to 1000 pages/second (`fastscan`) as it tends to 32 K (`minfree`). If after going twice through the whole of memory there is still a shortage, the swapper is invoked to swap out entire processes. The algorithm limits the number of pages scheduled to be paged out to 40 per second (`maxpgio`) since this is a good figure for the number of random I/O's 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.

Kernel Variables To Control Paging

minfree

This is the absolute minimum memory level that can be tolerated by the system. If (freemem - deficit) is less than minfree the system will swap processes out rather than paging. It is usually set to 32Kb and clamped at desfree/2.

desfree

This is the desired level of free memory that the pager will try to maintain. It is usually set to 100Kb and is clamped at (total memory)/16.

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 256Kb and is clamped at (total memory)/8.

fastscan

This is the number of pages scanned per second by the algorithm when there is less than minfree available. There is a linear ramp up from slowscan to fastscan as free memory goes from desfree to minfree.

slowscan

This is the number of pages scanned per second by the algorithm when there is desfree available. There is a linear ramp up from zero to slowscan as free memory goes from lotsfree to desfree.

maxpagio

This is the maximum number of page 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 disk. It can be increased if more or faster disks are being used for the swap space.

Context Switching

Table 5-2 shows the number of hardware contexts built into each machine. A hardware context can be though of as a tag on each PMEG entry in the MMU which indicates which process that translation is valid for. This allows the MMU to keep track of the mappings for 8, 16 or 64 processes in the MMU, depending upon the machine. When a context switch occurs, if the new process is assigned to one of the hardware contexts, then some of its mappings may still be in the MMU and a very fast context switch can take place. For up to the number of hardware contexts available this scheme is more efficient than a more conventional TLB based MMU. When the number of processes trying to run exceeds the number of hardware contexts the kernel has to choose one of the hardware contexts to be reused and has to invalidate all the PMEGS for that context and load some PMEGS for the new context. The context switch time starts to degrade gradually and probably becomes worse than a TLB based system when there are more than twice as many active processes as contexts. This is one reason why a SPARCserver 490 can handle many more users in a timesharing environment than the (apparently faster) SPARCstation 2, which would spend more time in the kernel shuffling the PMEGS in and out of its MMU.

How many nfsd's?

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. On a busy NFS server all the `nfsd` processes could be active. One optimisation is to increase the number of `nfsd` processes but care should be taken not to exceed the number of contexts in the MMU (or the capability of the disk I/O subsystem). On SPARCsystem 400 machines with 64 contexts in hardware the number can be increased from the default of 8 to 40 or more for a dedicated NFS server. On SPARCstation 1's and other machines with 8 contexts in hardware, the number should probably be reduced from 8 to something in the range 2 up to 6.

Configuring out unused devices

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 SLC'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 SLC) which can free up over 500Kbytes. The DL60 configuration file with all framebuffer types except mono removed and with TMPFS added makes a good DL20 kernel for the SLC.

References

- Building and Debugging SunOS Kernels, by Hal Stern
- SPARCstation 2 Performance Brief
- SunOS System Internals Course notes.
- System Performance Tuning, by Mike Loukides

Memory

6

This layer varies depending on the underlying hardware. Some performance tuning problems can be traced to the memory system. Unless you can choose a different machine to run on an algorithm change may be called for to work around the problem. This section provides information on the various SPARC platforms.

Cache Tutorial

Why Have a Cache?

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

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

Unfortunately, although main memory (known as DRAM) has increased in size, its speed has only increased a little. Today's processors are around ten times the speed of a Sun3/60 since they 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 130ns. The DRAM manufacturers have added special modes that allow fast access to a block of consecutive locations after an initial address has been loaded. These *fast page mode* DRAM chips can provide a block of data at the 25ns rate required for the SPARCstation 2 after a short delay to load up an address.

The CPU cache is an area of fast memory known as 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 in single cycle reads and writes and it transfers data from main DRAM memory using fast 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.

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 continues with no delay. If the data needs to be fetched from main memory then the CPU clock is effectively stopped for 26 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 26 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.

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. The effect of changes in hit rate for a 26 cycle miss cost are shown in table 6-1.

Table 6-1 Application speed changes as hit rate varies with a 26 cycle miss cost

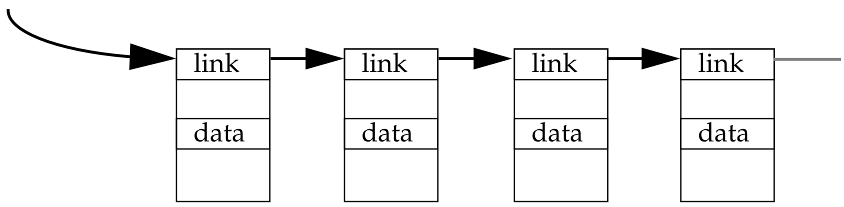
Hit Rate	Hit Time	Miss Time	Total Time
100%	100%	0%	100%
99%	99%	26%	125%
98%	98%	52%	150%
95%	95%	130%	225%

There is a dramatic increase in execution time as the hit rate drops. Although a 95% hit rate sounds quite high you can see that a 125% performance hit would be the result. Many small benchmarks like Dhrystone run at 100% hit rate. Many real programs have good data and code locality on average so they have a better than 99% hit rate. It isn't that difficult to do things that are bad for the cache however so it is a common cause of performance problems.

A Sample Problem

In chapter 2, on algorithms, I mentioned a CAD application that traversed a large linked list of 5000 entries. Each block on the list contains some data and a link to the next block. If we assume that the link is located at the start of the block and that the data is in the middle of a 100 byte block than the effect on the memory system of chaining down the list can be deduced.

Figure 6-1 Linked List Example



The code to perform the search is a tight loop shown in figure 6-2. 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 26 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.

Figure 6-2 Linked List Search Code in C

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

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

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

Figure 6-3 Linked List Search Loop in Assembler

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

Cache miss cost and hit rates for different machines

Since the hardware details vary from one machine implementation to another and the details are sometimes hard to obtain, the cache architectures of some common machines are described below, divided into four main groups.

Virtual Write Through Caches

Example Machines

The 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 and width (32 or 64 bits) of the buffer vary. If a number of words are written back-to-back then the buffer may fill up and the processor will stall until the slow memory cycle has completed. The processor waits until the entire cache block has been loaded before continuing. The SBus used on the SS2, IPX and ELC machines runs at half the CPU clock rate.

Table 6-2 Virtual Write Through Cache Details

Machine	Size	Line	Miss Cost	Depth	Width	Clock
SS1, SLC	64Kb	16 bytes	13 cycles	1 word	32bits	20MHz
SS1+, IPC	64Kb	16 bytes	13 cycles	1 word	32bits	25MHz
SPARCsystem 300	128Kb	16 bytes	18 cycles	1 word	64bits	25MHz
ELC	64Kb	32 bytes	26 cycles	2 words	64bits	33MHz
SS2, IPX	64Kb	32 bytes	26 cycles	2 words	64bits	40MHz

Virtual Write Back Caches

Example Machines

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

How It Works

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

Table 6-3 Virtual Write Back Cache Details

Machine	Size	Line	Miss Cost	CPU Clock Rate
Sun4/200 series	128Kb	16 bytes	7 cycles	16 MHz
SPARCsystem 400	128Kb	32 bytes	12 cycles	33 MHz

Physical Write Back Caches

Example Machines

SPARC compatibles from Solbourne and ICL use this cache type.

How It Works

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

KBus backplane on Solbourne systems and the HSPbus on ICL systems are 64 bits wide and run at about 16 MHz with a read throughput quoted by ICL of 66Mb/s.

Table 6-4 Physical Write Back Cache Details

Machine	Size	Line	Miss Cost	CPU Clock Rate
Solbourne S5	128Kb	32 bytes	18 cycles	33 MHz
Solbourne S5E	128Kb	32 bytes	21 cycles	40 MHz
ICL DRS6000/40	128Kb	32 bytes	14 cycles	25 MHz
ICL DRS6000/50	128Kb	32 bytes	18 cycles	33 MHz

On-Chip Caches

Example Machines

Highly integrated SPARC chipsets like the Matsushita MN10501 used in the Solbourne S4000 the Texas Instruments SuperSPARC (Viking) and the Fujitsu MB86930 (SPARClite) use this cache type.

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 tend to be *associative* in that a line can exist in the cache in several possible locations. If there are four possible locations for a line then the cache is known as a *four way set associative cache.* It is hard to build off chip caches that are associative so they tend to be *direct mapped*, where each memory location maps directly to a single cache line.

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

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

Table 6-5 On-Chip Cache Details

Processor	I-Size	I-line	I-Assoc	D-Size	D-line	D-Assoc
MB86930	2Kb	16	2	2Kb	16	2
MN10501	6Kb	?	?	2Kb	?	?
SuperSPARC	20Kb	32	5	16Kb	32	4

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. 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. For I/O intensive applications the SPARCsystem 400 often performs much better than its MIPS ratings would suggest.

References

- Sun Systems and their Caches
- A Cached System Architecture Dedicated for the System IO Activity on a CPU Board

Windows and Graphics

7

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

- OpenWindows Version2 White Papers
- SunPHIGS/SunGKS Technical White Paper
- XGL Graphics Library Technical White Paper
- Graphics Performance Technical White Paper
- SPARCstation 2GS / SPARCstation 2GT Technical White Paper

Disk

8

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 SCSI disk tuning performed for SunOS 4.1.1

As ever, the latest version of SunOS is the best tuned. In particular the SCSI disk driver 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 SCSI 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 quite close to its theoretical maximum.

If a SCSI 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.

Throughput of various common disks

Understanding the Specification

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

What the Makers Specify

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

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

What You have to Work Out for Yourself

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

Figure 8-1 /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 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 of the disk.

data rate (bytes/sec) = (nsect * 512 * 3600) / 60 = 1658880 bytes/sec

If we assume that 1 Mbyte is 0x100000 bytes then this is 1.58 Mbytes/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 1.58 Mbytes/sec for the minimum transfer size of 4 or 8Kb. Raw reads on this type of disk tend to run at just over 1.5 Mbytes/sec which confirms the calculated result. Some common Sun disks are listed in table 8-1.

Table 8-1 Disk Specifications

Disk	Formatted Capacity	Peak	Calculated	RPM	Seek
Quantum 105S	104.7 Mb	1.2	1.07 Mb/s	3600	20ms
Sun0207 SCSI	207 Mb	1.6	1.08 Mb/s	3600	16ms
Sun 0424 SCSI	424 Mb	3.0	2.80 Mb/s	4500	14ms
Sun 0669 SCSI	669 Mb	1.8	1.68 Mb/s	3600	16ms
Sun1.3G SCSI/IPI	1300 Mb	4.5	4.00 Mb/s	5400	11.5ms
SMD	688 Mb	2.4	2.01 Mb/s	3600	16 ms
SMD	892 Mb	2.4	2.01 Mb/s	3600	15 ms
CDC IPI 9720	1020 Mb	3.0	2.46 Mb/s	3600	16 ms
CDC IPI 9722	911 Mb	6.0	4.68 Mb/s	3600	15 ms

Effect of disk controller, SCSI, SMD, IPI

SCSI Controllers

SCSI Controllers can be divided into two generic classes, those that support *Synchronous SCSI* operation and those that support *Asynchronous SCSI.* Working out which type you have on your Sun is not that simple. The main difference between the two is in the number of disks that can be supported on a SCSI bus before the bus becomes saturated.

A SCSI bus can transfer data at 5 or 6 Mb/s at its best. 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 much 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 with SunOS 4.1.1 and is usually 3.5 to 4.5 Mb/s.

The original SPARCstation 1 and the VME hosted SCSI controller used by the SPARCserver 470 do not support Synchronous SCSI. In the case of the SPARCstation 1 it is due to SCSI bus noise problems that were solved for the SPARCstation 1+ and subsequent machines. If noise occurs during a Synchronous SCSI transfer a SCSI reset happens and, while disks will retry, tapes will abort. In versions of SunOS before SunOS 4.1.1 the SPARCstation 1+, IPC, SLC have Synchronous SCSI disabled.

The 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 and the SPARCstation 330 are also capable of this speed. Other SCSI interfaces are not able to drive the SCSI bus at full speed and do not perform as well if three or four disks are active on a single SCSI bus.

SMD Controllers

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

IPI Controllers

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

Load Monitoring and Balancing

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

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

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

The second form of `iostat` was introduced in SunOS 4.1. The above output shows paging activity on a SPARCstation 1 with Quantum 104Mb disks, the SCSI device driver does not collect seek information so the `msps` field is always zero. It can be used with SMD and IPI disks to see whether the disk activity is largely sequential (less than the rated seek time) or largely random (equal or greater than the rated seek time). The first line of output shows the average activity since the last reboot, like `vmstat`.

Mirrored Disks

Sun's SPARC Server Manager 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.

CPU

9

This chapter looks at some SPARC implementations to see what differences exist.

Architecture and Implementation

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

Instruction Set Architecture (ISA)

Defined by the SPARC Architecture Manual, Sun has published Version 7 and Version 8, the IEEE have a draft standard based on Version 8, IEEE P1754.

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.

Kernel Architecture

A number of similar systems may be parameterised so that a single GENERIC kernel image can be run on them. This grouping is known as a kernel architecture and Sun has one for VME based SPARC machines (Sun4), one for SBus based SPARC machines (Sun4c) and one for the VME and SBus combined eurocard machines (sun4e).

The effect of register windows and different SPARC CPU's

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

- SPARC Architecture Manual Version 8
- SPARC Strategy and Technology

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 stack. 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 around 35% and provide a speed up of 5-15%. Care must be taken to avoid writing code that makes a large number of recursive or deeply nested calls, and keeps returning to the top level. If very little work is done at each level and very few parameters are being passed the program may generate a large number of save and restore traps. The SPARC optimiser can perform tail recursion elimination and leaf routine optimisation to reduce the depth of the calls.

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 tens of microseconds on average. 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 this ratio of 100 subroutine calls per context switch is a good one since there is time to take advantage of the register windows before they are flushed again. When the new context starts up it loads in the register windows one at a time, so programs that do not make many subroutine calls do not load registers that they will not need. Note that a special trap is provided that can be called to flush the register windows, this is needed if you wish to switch to a different stack as part of a user written co-routine or threads library. When running SunOS a context switch rate of 1000/second is considered fast so there are rarely any problems. An unbuffered serial port running at 9600 baud will generate interrupts at about 1000/second so this also looks OK. There may be more concern about this problem when running real time operating systems on SPARC systems, but there are alternative ways of configuring the register windows that are more suitable for real time systems. These are discussed in the following reference.

- SPARC Technology Conference notes - Alternative Register Window Schemes

How many cycles each instruction takes on different SPARC CPU's

Most SPARC Instructions execute in a single cycle. The main exceptions are floating point operations, loads, stores and a few specialised instructions. The time taken to complete each floating point operation in cycles is shown in table 9-1 for the Weitek 3170 used in the SPARCstation 1, 1+, SLC and IPC machines and the Texas Instruments 8847 used in the SPARCsystem 300 and 400 machines. The SPARCstation 2 uses an improved version of the TI8847 and the SPARCstation ELC and IPX use an improved version of the Weitek 3170.

One factor that is not apparent from the table is that these devices have a floating point instruction queue, that is they can start new instructions before the old ones have finished. The TI8847 has an extra stage in its queue which allows it to have more things happening at once but delays the result. In practice performance on most multiply/add/subtract intensive code is the same on both systems. If there are lots of square roots and divides then the TI8847 is much faster. This explains the identical Linpack figures for the SPARCstation 2 and IPX, since Linpack is multiply and add intensive. The IPX is a little slower on SPECmarks since there are some square roots and divide instructions in the SPEC benchmark suite.

The FPS 500EA uses the BIT B5000 unit which is an ECL SPARC implementation announced in 1989 that has been optimised for high performance floating point operations.

The next generation SPARC chips from TI (SuperSPARC/Viking) and LSI (Lightning) have not had full details disclosed but these are both superscalar processors which can execute several instructions in a single cycle. The SuperSPARC can execute two integer and one floating point instruction every cycle and just about the only instructions that take longer are integer multiply and divide, and floating point divide and square root.

The integer units also vary in the time taken to perform loads and stores, this is shown in table 9-2 for the Fujitsu/LSI implementation used in the original Sun4 and SPARCstation 1, 1+, SLC and IPC machines; the Cypress implementation used in the SPARCsystem 300, 400 and SPARCstation 2, ELC and IPX; the BIT B5000 used by FPS; and the Fujitsu SPARClite, a recent development aimed at the embedded control marketplace.

Table 9-1 Floating point Cycles per Instruction

Instruction	Weitek 3170	TI8847	BIT B5000
fitos	10	8	2
fitod	5	8	2
fstoir	5	8	2
fdtoir	5	8	2
fstoi	5	8	2
fdtoi	5	8	2
fstod	5	8	2
fdtos	5	8	2
fmovs	3	8	2
fnegs	3	8	2
fabss	3	8	2
fsqrts	60	15	24
fsqrtd	118	22	45
fadds	5	8	2
faddd	5	8	2
fsubs	5	8	2
fsubd	5	8	2
fmuls	5	8	3
fmuld	8	9	4
fdivs	38	13	14
fdivd	66	18	24
fcmps	3	8	2
fcmpd	3	8	2
fcmpes	3	8	2
fcmped	3	8	2

Table 9-2 Number of Register Windows and Integer Cycles per Instruction

Instruction	Fujitsu/LSI	Cypress	BIT B5000	SPARClite
(register windows)	7	8	7	8
ld (32 bit integer)	2	2	1	1
ldd (64 bit integer)	3	3	2	2
ld (32 bit float)	2	2	1	1
ldd (64 bit double)	3	3	1	2
st (32 bit integer)	3	3	2	1
std (64 bit integer)	4	4	3	2
st (32 bit float)	3	3	1	1
std (64 bit double)	4	4	1	2
taken branches	1	1	0	1
untaken branches	2	1	1	1
jmpl	2	2	2	2

Network

10

This topic has been extensively covered in other white papers. An overview of the references is provided here.

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.

When routing packets from one ethernet to another a SPARCstation 1 running SunOS 4.0.3 could route up to 1815 packets/second. 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.

- An Analysis of TCP Processing Overhead, IEEE Communications

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 so heavy disk activity can reduce ethernet throughput. The add-on SBus ethernet cards use exactly the same interface as the built-in ethernet, but have an SBus DMA controller to themselves. These machines give very good ethernet routing performance and can be configured with a total of four ethernet interfaces if no framebuffer is present.

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 a bit 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 - nc

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. There are no performance figures for routing throughput.

Routing Throughput

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

Table 10-1 Ethernet Routing Performance

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

Using NFS effectively

See the following references:

- Networks and File Servers: A Performance Tuning Guide.
- SPARCserver 490 NFS File Server Performance Brief
- Tuning the SPARCserver 490 for Optimal NFS Performance
- SunOS 4.1 Performance Tuning.

References

A

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

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

Building and Debugging SunOS Kernels, by Hal Stern

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

A 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.

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 SCSI 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.

Networks and File Servers: A Performance Tuning Guide, December 1990

This is right up to date and is the definitive document for network tuning.

OpenWindows Version 2 White Papers

This includes `x11perf` results for OpenWindows 2 compared with MIT X11R4 and DECwindows.

Optimisation in the SPARCompilers, Steven Muchnick

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

SBus Specification Rev B

This is the latest version of the SBus specification, including 64bit extensions. It is available free from Sun sales offices.

The SPARC Architecture Manual Version 8

This will be published in book form during summer 1991. It is also available from the IEEE P1754 working group.

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.

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.

SPARC Technology Conference notes - 3 Intense Days of Sun

The SPARC Technology Conference has been touring the world for the last year or more. It is particularly aimed at SPARC hardware and real time system developers. You need to go along to get the notes!

SPARCserver 490 NFS File Server Performance Brief, February 1991

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

SPARCstation 2GS / SPARCstation 2GT Technical White Paper

This describes the highest performance graphics accelerators available from Sun.

SPARCstation 2 Performance Brief

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

SPARCserver Performance Brief

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

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. It is available from Sun.

SunOS System Internals Course Notes

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

SunPHIGS / SunGKS Technical White Paper

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

System Performance Tuning, by Mike Loukides, O'Reilly, 1991.

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

tmpfs: A Virtual Memory File System, by Peter Snyder

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

Tuning the SPARCserver 490 for Optimal NFS Performance, February 1991

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

XGL Graphics Library Technical White Paper

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

You and Your Compiler, by Keith Bierman

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

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

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

European Headquarters
Sun Microsystems Europe, Inc.
Bagshot Manor, Green Lane
Bagshot, Surrey GU19 5NL
England
0276 51440
FAX 0276 51287

Australia: (02) 413 2666
Belgium: 32-2-759 5925
Canada: 416 477-6745
Finland: 358-0-5022700
France: (1) 30 67 50 00
Germany: (0)89-46 00 8-0
Hong Kong: 852 5-8651688
Italy: 039 60551
Japan: (03) 221-7021
Korea: 2-563-8700
The Netherlands: 033 501234
New Zealand: (04) 499 2344
Nordic Countries: +46 (0)8 705 30 00
PRC: 1-8315568
Singapore: 224 3388
Spain: (91) 5551648
Switzerland: (1) 825 71 11
Taiwan: 2-7213257
UK: 0276 20444

Europe, Middle East, and Africa,
call European Headquarters:
0276 51440

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