

Demystifying High-End Sun Fire Behavior when Scaling Database Applications

Glenn P. Fawcett
Marcus Heckel
Sun Microsystems, Inc.

Abstract

When business grows, so does the IT infrastructure. With special promotions and the holiday shopping, business volume can experience an order of magnitude growth. Planning for anticipated growth is essential to business - if the servers cannot handle the load, revenue will be lost.

We often see the simple approach of “buying” your way through a peak period – adding CPU, memory, and IO to get the business through the peak. This approach can lead to disaster if the software stack is not considered and expectations are not properly set.

Scaling of a complex multi-tier application requires proper attention to the application, OS, network, IO subsystem, memory hierarchy, and CPU count all at the same time. When switching from a mid-tier server to a high-end server, architectural differences as well as just the sheer number of CPUs will cause the machine to look and feel different.

The end goal of migrating from a 24-core SF6800 to a 144-core E25k is to achieve a higher level of throughput. While system statistics might be different, the overall system throughput, while meeting response time requirements, is what matters.

Many books have been written on application scaling, database tuning, system tuning, and networking. This paper is not going to revisit these topics in detail, but will focus on topics which aid or hinder scaling on the largest of Sun Fire servers. With an increased understanding of system

behavior, customers should be able to better scale their applications on Sun Fire servers.

Architectural Choices

Computer architectures have been evolving for many years. As noted in Alan Charlesworth's paper¹ on the Fire Plane interconnect, the success of an SMP system interconnect lies in the efficiently delivering application performance and throughput from the processors, disk drives, and DIMMS. Multiple levels of interconnect are required to achieve high bandwidth on large SMP servers, forcing a design trade-off of higher latency. It is the natural tendency to compare servers up and down the product line, but design goals must be considered in-order to understand performance.

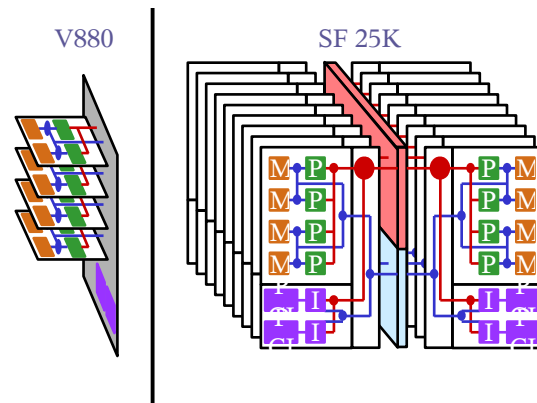


Figure 1: Comparison of Architectures

Comparing a V890 server to a E25K makes about as much sense as comparing a Porsche to a Semi-Truck. The Porsche will

¹Charlesworth 2001 “The Sun Fireplane System Interconnect”

certainly get around the track faster than the Semi-Truck but it would have a difficult time carrying much more than two passengers and an overnight bag. These two vehicles are designed for different purposes. When scaling applications across architectures, it is important to keep in mind the design goals so as to better understand the real architectural differences.

<i>Server</i>	<i>#cpu</i>	<i>BW GB/sec</i>	<i>Local Latency (ns)</i>	<i>Remote Latency (ns)</i>
E25K	72	76.1	248	468
E6900	24	9.4	221	268
V880 @900MHz	8	3.3	-na-	252
V280R @900MHz	2	1.3	-na-	228

Real Architectural Differences

So what are the real differences between the Sun Fire E25K and smaller servers?

- **Memory Size.** Current Sun Fire Servers can handle about 8GB/CPU. On a V440 with 4 CPUs this totals 32GB, whereas a E25K can have up to 576GB of memory. The larger memory is needed to keep a balance and fully utilize the CPUs on these servers.
- **Memory Bandwidth and Latency.** The Sun Fire E25K has memory spread across all of the 18 System boards. This hierarchy is necessary to allow the system to scale to over ½ Terabyte of memory and a total system bandwidth of 76 GB/sec. The E25K can accommodate 3x the number of processors and more than 8x the memory bandwidth than the Sun Fire E6900. But, along with the hierarchy comes increased latency. Table 1 shows the bandwidth and latency of various Sun Fire servers².

Table 1 - Memory Bandwidth and Latencies

- **IO Throughput.** There are big differences in IO throughput as server sizes increase. A V280R might be able to deliver better latency numbers in terms of IO and memory access, but throughput is far less than the Sun Fire E25K. An E25K has been shown to deliver over 18GB/sec of IO. This makes the E25K well-suited for decision support applications where large amounts of data is accessed. In fact, over the history of the Sun Fire E15K/E25K line, there have been numerous world records on the TPC-H³ benchmark which depends on IO bandwidth for DSS performance.
- **Scheduler Dispatch Table.** By default, the E15/25K has a different dispatch table than smaller servers. The dispatch table determines the time-slice of processes at various priority levels. A smaller time-slice is consistent with more interactive workloads like web , whereas a larger time-slice is used more for database processes and batch workloads. Each unique workload may respond differently to this parameter.
- **Kernel Cage.** To improve uptime by enabling Dynamic Recovery, the kernel cage is enabled on E6900/25K systems. The kernel cage corrals memory used by the kernel on one or two system boards. This allows system boards to be

²Lmbench and Streams benchmarks were used to measure latency and bandwidth.

³ See <http://www.tpc.org/> for results.

configured and removed without shutting down the domain. Smaller Sun Fire servers don't enable the cage since on-line board swap is not possible. In cases where network IO is extreme, disabling the kernel cage may improve performance. Beware, that this will disable on-line board configuration so there are trade-offs to be considered.

Misunderstood System Statistics

There are several statistics which are often misunderstood. Some of these statistics can be used to characterize performance and aid in growing applications, while others are completely worthless. This section will discuss the usefulness of each statistic and how it may react with system growth.

- **Wait IO: (WaitIO = IDLE!!)**

This statistic is most commonly seen in `vmstat(1M)`, `iostat(1M)`, and `mpstat(1M)`. "Wait IO" is also available via `kstat(1M)` so unfortunately it shows up in third party tools like Patrol and TeamQuest. So, why does this statistic lead the performance analyst astray?

"Wait IO" is really nothing more than a measure of idle on modern systems. Every time a CPU receives a clock tick either `CPU_IDLE`, `CPU_WAIT`, `CPU_KERNEL`, or `CPU_USER` are incremented depending on the state of the CPU at the time of the tick. If the CPU is idle either `CPU_IDLE` or `CPU_WAIT` is incremented. When an IO is queued and a process goes to sleep waiting for IO it calls `biowait()` and this increments a per CPU data structure. If there is a lot of IO occurring on a system, idle time will tend to be accounted for under `CPU_WAIT`. This statistic had "some" meaning when we had single CPU systems that did synchronous IO. As soon as async IO and multi-cpu systems were common, this statistic was misleading.

It is natural to see more "Wait IO" on a larger system simply because it can do more IO. High Wait IO is merely an indication that there is IO occurring, it does not indicate a problem. That is not to say that there is not an IO problem, merely that Wait IO alone is not sufficient to determine a problem. To diagnose IO problems, use `iostat(1M)` to analyze the response time, active queue, and throughput of the device.

There is good news on the horizon for the system performance analyst. Solaris 10 `CPU_WAIT` will NOT be incremented and `CPU_IDLE` will get updated instead. This will cause "Wait IO" to show up as ZERO on all Solaris 10 systems.

- **Cross Calls "xcal"**

Cross calls can be seen using the `mpstat(1m)` command. An increase in cross calls can cause alarm for system administrators, and system performance can come into question. So what are these mysterious entities that raise the blood pressure of system administrators?

A cross call is used by the kernel to instruct a specific processor to execute a low-level function. Inter-processor interrupts and virtual memory translation consistency are implemented as cross calls⁴. These include things like: signal processing, dispatcher preemption, `"/proc"` thread control, and virtual memory (VM) consistency. It stands to reason that on a system with more processors and a higher level of throughput, cross calls would also increase.

Of all the reasons for cross calls, VM consistency is the most common. Each CPU has TLB entries which map memory from the processors MMU to real memory. When pages are unmapped from a user's address space these entries must be invalidated on all the CPUs the process has run. When a process exits the entire address

⁴Solaris Internals, Mauro & McDougall,
©2001 Sun Microsystems ISBN 0-13-022496-0

space for that process is unmapped. This can lead to what we call an exit storm which results in a flurry of cross calls. Most applications have ways of maintaining persistent connections which limit the cross-call storm caused by address space tear down.

Additionally, applications which use buffered file-systems will see a large increase in cross calls due to the way the page cache is managed with segment maps. This will be covered in more detail later in this paper.

Finally, to further analyze cross-calls on Solaris 10, the following `dtrace(1M)` command can be run.

```
dtrace -n 'xcalls{@a[stack(20)] = count()}'
```

Mastering Growth

Applications are made up of many layers of HW and SW. In order to master the growth of an application, every layer must be in balance. Focusing on one layer but ignoring others is a recipe for disaster.

Most IT systems can be described as layers consisting of the:

- User Application
- Database
- Operating System
- Firmware
- Hardware

Most classic tuning texts focus on the application layer first. The application is definitely the BEST place to make changes, but it is often the hardest. Additionally, if the rest of the stack is not up to par, performance issues will be experienced in the best-designed applications. Let's turn the stack upside down and see what can be done without a re-design.

It is still possible that after going through the steps and building the application environment that it may be inevitable to redo the application, but that is a whole other topic.

The remainder of this paper will focus on topics which allow the reader to master their environment so they can build an environment which performs and scales well.

1. Hardware Choices

One of the most simple, but often overlooked pieces of performance is hardware. It is the underlying foundation of whatever operating system or application which is being run. At this point in the product life-cycle of the Sun Fire 15K and E25K, there have been numerous changes to its underlying structure. UltraSPARC IV, ASICs, PCI bridge chips, host bus adapters, and firmware levels have all undergone changes.

When Sun Fire 15K first shipped, the concept of Memory Placement Optimization (MPO, described in OS choices) was still in development. Initially, the AXQ ASICs were unable to fully take advantage of MPO optimizations available in Solaris 9. The AXQ 6.2 ASIC was enhanced to take advantage of MPO and fully utilize the new HPCI+ assembly now shipping with Sun Fire 15K and E25K⁵.

The HPCI+ assembly allows full bandwidth to be utilized via the host bus adapters now supported on these platforms. The HPCI+ assembly itself has 1 more 66MHz slot and is capable of almost double the throughput than the previous HPCI generation. Sun Fire 15K and E25K systems shipped today contain latest AXQ ASICs, systems while older expander boards may not. Running the right hardware and operating system to

⁵The AQX version can be found using the "redx" command on the service processor. Contact a service representative for assistance.

take advantage of MPO is critical to scalability.

2. OS choices Solaris 8/9/10

One of the most obvious factors when upgrading from a work-group or mid-range server to a high-end server, is the memory and cache coherency differences. Smaller machines typically have less latency to memory simply because there are less processors which must maintain coherency.

The Sun Fire 15K and 25K have a memory crossbar which connects local memory to remote memory on other boards. This hierarchy of interconnect, while transparent to the user, is necessary to scale large servers. While this architecture is transparent to user applications, to take full advantage of the server, special considerations must be made. The most basic consideration is to choose an operating system that matches the memory architecture.

When Solaris 8 was first developed, machines with a hierarchal interconnect where not available⁶. The Solaris development team began to focus on large memory optimizations with Solaris 9 and Memory Placement Optimization (MPO) was born. This was a radical departure from the uniform memory structure of the past. The memory subsystem was split into pools on each board where multiple free lists are maintained and processes have affinity to run on their locality group (lgroup). MPO increases the probability of a processes using local memory which can increase performance.

Because of the increased memory efficiency and numerous other factors, you should plan on upgrading the operating system as part of any migration strategy.

Organizations often put up road blocks to migrating to the latest OS release. Certification and re-testing is often hotly debated among application vendors, OS vendors, and IS departments. Solaris has a application guarantee which states that we will certify that your application will run across OS releases⁷.

3. Memory page sizes

TLBs are used to map virtual to physical memory. As memory sizes grow, more mappings are required. UltraSPARC processors cache a limited number of memory pages based on page size. If the TLB doesn't reside in the processors cache a TLB miss will occur. The time spent processing TLBs of various page sizes can be seen using the `trapstat(1M)` command:

```
# trapstat -T
```

High miss rates can occur when a large amount of small pages are used. Consider an 8K memory page versus a 4M. To map a 4M memory page, one TLB entry is needed. To map the same amount of memory (4MB) with 8K pages, 512 TLB entries are required. On a machine with over ½ a Terabyte, TLB processing with small pages can reduce performance.

Solaris 9/10 allows for 8K, 64K, 128K, and 4M pages. Earlier versions allowed for 4M to be associated with ISM shared memory segments only. Solaris 9 introduced the ability to control memory sizes of the heap and stack of user processes. This can be done via an API or by using the `LD_PRELOAD` function to load the `mpss.so` library.

As mentioned before, there is also an API where the developers can decide the sizes of the memory pages. Oracle has recently shown good gains by using large pages for

⁶Solaris 8 is due to be obsolesced in Oct 2005.

⁷<http://www.sun.com/software/solaris/p/programs/guarantee.xml>

anonymous mmap() memory. This can be turned on in Oracle 10G by using the:

```
_use_realfree_heap_pagesize_hint=4194304
```

init.ora parameter. This improved query performance on a recent benchmark by 14%. The dTLB misses were reduced by 12% and the user/system time went from 60%/40% to 90%/10%.

4. Memory Fragmentation

File systems are often deployed for database files. System administrators often prefer this method to managing links and raw data files. Even when deploying file systems you have many options, namely whether or not a file is buffered.

Environments which have 32-bit databases have got in the habit of using file systems to help improve IO performance. A smaller memory system might be running just fine with this configuration, but suffer when a server upgrade occurs and memory is substantially increased. Indeed there are also some situations where a 64-bit database might benefit from double buffering, but either way, this will cause more overhead than buffering within the database itself.

One customer situation occurred where they were migrating from a 48GB SF6800 to a 288GB E25K. After migrating the database, start times were dramatically increased. This increase was related to memory fragmentation. When database files are placed on a buffered file system, IO is buffered in 8KB chunks and mapped to 8KB memory pages. Since these are database files, IO is constantly occurring causing Solaris to reserve more and more 8KB pages for the file system until eventually all memory is used. This memory can easily be reused as 8KB pages, but must be coalesced before being used for 4MB pages. This is where the fragmentation occurs.

Oracle uses ISM or DISM which consists of 4MB pages. This improves efficiency by lowering the dTLB overhead. If there are not enough 4MB pages, memory coalescing kicks in and attempts to reconstruct 4MB pages out of 8KB pages. This is a very expensive operation on Solaris 8 with one memory free list. Other processes are get starved for memory and the system seems to grind to a halt. On Solaris 9 and 10 things are better due to multiple free lists and improved coalescing efficiency.

Buffered IO is just one of the causes of memory fragmentation. To be complete they are:

- Crash dumps. Crash dumps write large core files the crash directory. Since this is typically on a buffered file system memory can become fragmented.
- User applications which malloc large amounts of memory.
- VxFS with Cached QIO. Cached QuickIO is nothing more than buffering of IO in the page cache by Veritas.

Most of the causes are fairly easy to avoid:

- Mount UFS file systems with "forcedirectio"
- Disable VxFS Cached QIO. If you are not comfortable with going all the way on this one, you can selectively enable QIO on a per file basis. Use the qiostat utility to monitor which files are benefiting the most.
- Use 64bit Oracle to buffer data in the buffer cache instead of the file page cache.
- Upgrade to Solaris 9/10 and use 4MB pages for applications the allocate large amounts of memory.

5. Segmap scaling.

Solaris uses pre-built mappings of buffers to pages in the buffer cache. As the amount of memory in the page cache increases, less

pages are covered by maps and must be created on the fly. As the page cache grows memory becomes more fragmented into 8K pages and the likelihood of pre-built mappings is decreased. If the `segmap_percent` limit has been reached, then a valid mapping must be invalidated which uses additional system resources.

So, how do you see how much memory is actually being used by the page cache?

```
# mdb -k
Loading modules: [ unix krtld genunix ip usba wrsm random
ipc nfs ptm cpc ]
> ::memstat
Page Summary          Pages          MB  %Tot
-----
Kernel                430030          3359   2%
Anon                  805572          6293   3%
Exec and libs         9429             73   0%
Page cache            14974588        116988 52%
Free (cachelist)      2547680          19903  9%
Free (freelist)       9853807          76982 34%
Total                 28621106        223602
```

Figure 2 Monitoring the page cache

If you are on Solaris 8, you will need to install the `memtool`⁸ written by Richard McDougal. If you are on Solaris 9/10 you can use `mdb(1)` and use the `::memstat` command. Beware, this command will take a long time to run and may effect performance, therefore it is best to run this when the system is not busy.

After figuring out how much memory is used for the page cache, you can monitor the segmap hit percentage. This can be done via the `kstat` interface. A favorite method is to use the `segmapstat` command which is part of the `cachekit`⁹ freeware tools.

```
root# ./segmapstat 10 100
segm  %hit    hit      miss
-----
 97.438  17074    449
 98.654  20445    279
 74.444  26584    9126
 70.192  43137   18319
 65.825  37007   19213
 65.681  35616   18610
 71.178  46973   19021
 63.439  33521   19319
 56.428  25357   19580
```

Figure 3 Segmapstat sample output

If this percentage is consistently low (< 75%), consider increasing the parameter `segmap_percent`. By default, this is set to 12% of memory. When super caching large amounts of data in the page cache, the `segmap_percent` often has to be increased.

Reducing the amount of memory in the page cache is the best course of action. Caching data with 4MB pages in a 64bit database is far more efficient than using the page cache. Having said that, there are circumstances where performance can be increased by using the page cache.

Monitoring the size of the page cache and the segmap hits can help you arrive at a more appropriate value for the `segmap_percent`.

6. Buffered IO vs DB caching

Using a buffered file system (VxFS, UFS, or Cached QIO) has distinct differences to using a 64bit version of the database to cache database blocks. A buffered file system uses more resources when retrieving data than the database buffer cache. A system with buffered IO on a file system will show more xcalls (cross-calls), system CPU utilization, and segmap operations. This overhead will increase with the number of cores and reduce overall throughput. Consider the following test case.

Setup:

- Oracle 9iR2 64bit
- create a 46GB table.
- Spawn 100 users to select 1/100 of the table with singleton selects.

This test was run on both buffered and direct mounted file systems. The `segmap_pct` was left default at 12% which equates to 23GB of segmaps. Table 2 below shows results on a Sun Fire E6900 with 192GB of memory and 24

UltraSPARC IV processors running at 1200MHz.

<i>Cache</i>	<i>OS</i>	<i>Xcall/sec</i>	<i>rtime</i>	<i>Usr</i>	<i>sys</i>
FS	S9	2,600,000	63	71	28
DB	S9	3,000	28	94	5
FS	S10	500	54	77	21

Table 2: DB vs FS caching efficiency

The results are pretty interesting. Run times were cut by more than half when going from FS caching to DB caching. We see a huge reduction in cross-calls (xcall) as well as system CPU time. This example shows that it is vastly more efficient (50%+) to cache in the database verses the file page cache. One interesting note is the Solaris 10 result with FS caching.

Solaris 10 eliminates the cross-calls associated with the file system caching. This equates to a 16% improvement in overall runtime. While this improvement is admirable, it is still a long ways from the efficiency gains seen by DB caching.

7. Sizing up Latency effects

High-End Sun Fire Servers have more latency than entry-level servers due the multiple layers of interconnect required for scaling. But to what degree does an increase in latency effect performance?

A unique way to look at latency effects is by examination of `cpustat(1M)` counters. Use `cpustat(1M)` to find the ratio of number of cycles spent waiting for caches to be filled to total_cycles with the following `cpustat(1M)` command:

```
# cpustat -c pic0=Cycle_cnt,pic1=Re_EC_miss,sys 5 2
2.007 4 total 7212880624 780492504
```

This result shows 10.8% of time spent in latency based waits. Ratios under 20% for pic1/pic0 would represent a case where the F15K/E25K is not spending enough time in latency induced wait states to have an effect

on overall system performance. Ratio's above that threshold would indicate potentially worthwhile improvements from MPO tuning.

A high number here is not necessarily bad by itself. It is also characteristic of the idle loop, where most of the time is spent waiting on a lock protecting the run queues. Unfortunately, this `cpustat(1M)` command doesn't work very accurately for systems with a lot of idle. Some workarounds for this situation are:

- If the application doesn't spend much time at all in system time, just ignore that. (i.e. don't use the ",sys" option in `cpustat(1M)`)
- Use `pbind` and only look at certain CPU if the application can peg the CPU at 100%.
- Use an idle soaker when running `cpustat(1M)`. A simple looping script that soaks up unused cycles.



8. Kernel cage considerations

The kernel cage was put into Solaris to enable hardware Dynamic Reconfiguration (DR). On the Sun Fire 15K and E25K, the cage will be placed on the highest board configured into the domain at boot time. To determine where the cage lives, utilize `cfgadm(1M)`.

```
> cfgadm -alv | grep -i permanent

SB17::memory connected   configured ok
base 0x22000000000, 8388608 KBytes
total, 5718416 KBytes permanent
```

The kernel cage contains crucial data structures used for operation of the kernel (managing synchronization, network buffers, interrupts, etc). When all of these structures are placed on one system board, this can create hot spots which wouldn't

occur on systems without a kernel cage. As system resources increase, the pressure exerted on these data structures causes increased contention and processes stall trying to manipulate these structures.

Determining if the kernel cage is an issue on a specific workload is fairly straightforward. Two performance counters, available via `cpustat(1M)`, can help diagnose cage issues. The `MC_reads_[0-3]` and `MC_writes_[0-3]` counters are much higher (could be 4x or more) compared to CPUs without the kernel cage. Disabling the kernel cage will result in memory access being much more uniform as critical data structures will be spread out over the entire system. Doing this can result in 5%, 15%, or even more performance gain depending on the pressure which was exerted on the cage.

The benefit gained from disabling the kernel cage must be weighed heavily against the loss of DR. In many circumstances, disabling DR is not an option.

The latest kernel patches to Solaris and the release of Solaris 10 have eased the pressure on the kernel cage. Specifically 5050686 “Solaris mutexes should be made more efficient under contention” and 5054052 “`disp_getwork()` is greedy and negatively impacts dispatch latency” have helped in this manner. Kernel cage enhancements continue to be developed within Solaris engineering.

9. Veritas configuration choices

VxFS is commonly used, but without the proper configuration, performance can suffer. Buffered VxFS or VxFS with cached QIO has characteristics similar to buffered UFS. Some performance gains can be realized, but you pay the price in terms of system overhead. As shown by the example earlier in this paper, you can get double the throughput by caching in the

DB as opposed to the file system. The following is a summary of Veritas configuration choices.

Good choices:

- VxFS ODM.
- VxFS QIO

Bad Choices:

- VxFS Buffered. Avoid this for the same reasons as buffered UFS.
- VxFS /w Cached QIO. This is really the same as VxFS or UFS Buffered, the only exception is IO is done using KAIO then copied to the page cache. If you must use Cached QIO, at least enable it selectively (ie, the redo log files).

A must have for ALL Veritas configurations:

- **Increase vxiod thread count.**

By default, Veritas enables only 10 vxiod threads. This was first seen as a problem on Starfire servers. You can check the value of this setting by using the “vxiod” command with no options. It can be modified on the fly, so don't be shy.

```
vxiod -set <# cpu cores>
```

This can be set in the `/etc/init.d/vxvm-startup2` and `/etc/init.d/vxvm-reconfig` files so that when the server restarts, the number of worker threads will be optimal. A good rule of thumb is to set it to the number of on-line cores¹⁰. Unfortunately, Veritas still has a max limit of 64 threads, so on machines with more than 64 cores, the worker threads will be limited.

10. Networking

Typically on the Sun Fire 15K and E25K , networks are used for large bulk transfers of data- like backup using (rdist, tar,

¹⁰UltraSparc IV processors contain 2 cores each

ufsdump, ftp, etc.). Unlike a v880, which has good out of the box performance, the E25K architectural differences sometimes require that we tune Solaris to get best performance. Latest out of the box measurements on the E25K show excellent performance. However, if not running with the latest updated hardware and Solaris 10, the following tunings have helped tests using the Cassini Ethernet driver with 1 Gigabit cards, obtaining results very close to those of the V880. Use the following tunings for large bulk transfers with low numbers of connections. For small packet sizes, no tuning is currently recommended.

```
Insert "interrupts=1;" > ce.conf under
/platform/sun4u/kernel/drv/
```

This will force the CE driver to interrupt 1 CPU. Furthermore, disabling the task queue in `/etc/system` will result in packets being processed in interrupt context.

```
set ce:ce_taskq_disable=1
```

Utilize MPO to try and keep memory access local to the lgroup of the servicing thread. To do this, set the following `/etc/system` parameter:

```
set lgrp_mem_default_policy=1
```

Finally, if the applications makes use of `setsockopt()`, the send/receive socket buffer watermarks can benefit from a larger buffer size. The following example shows how the socket buffer watermarks are adjusted.

```
ndd -set /dev/tcp tcp_recv_hiwat 1048576
ndd -set /dev/tcp tcp_xmit_hiwat 1048576
```

With these settings, the Sun Fire 15K was able to achieve 898 Mb/s stream out and 650 Mb/s stream in with over 5000 packets per second. Table 3, referenced below shows the E20K with Solaris 10 delivers great performance out of the box.

	<i>OS</i>	<i>Trans Mb/s</i>	<i>Rec Mb/s</i>
<i>15k</i>	<i>S9</i>	<i>646</i>	<i>345</i>
<i>15k tuned</i>	<i>S9</i>	<i>898</i>	<i>650</i>
<i>20k</i>	<i>S10</i>	<i>901</i>	<i>818</i>

Table 3: Networking speeds and feeds

11. Scheduling

The dispatch table is used by the kernel scheduler to decide which process gets to run when the CPUs get busy. From the classic point of view, CPU intensive processes are pushed to lower priority automatically by the fact that they are still ready to run when the time "Quanta" at the their level expires. Interactive performance is enhanced by letting the associated processes float to higher priority by noticing that they go to sleep before their quanta expires. This classic view is how the default Sun table works.

On the other hand, for big database servers, this results in a situation where a CPU intensive portion of the database is holding a data-structure lock as it is running at low priority. To avoid this, we use higher values for the time quanta. This may be known as the "Starfire" dispatch table or the "Cray" dispatch table. By default the larger Sun Fire Servers load a table with a larger time quanta. This larger time quanta may hurt performance if the system is used for more of an interactive workload.

To display and modify the current dispatch table, use the `dispadmin(1M)` command. To view the current dispatch table do:

```
dispadmin -c TS -g
```

Solaris 8, 9, and 10 all have the dispatch table. Starting with Solaris 9, a new Fixed priority (FX) class scheduler has been introduced.


The Fixed priority scheduler fixes the priorities and time-slices of individual processes. This allows an extra added level

of control over the classic dispatch table model. Processes are placed into the FX class via the `priocntl(1M)` command.

```
priocntl -s -c FX -m 58 \  
-p 58 -t 1000 -i pid
```

This has been useful for applications which mix batch and interactive performance. Processes which need interactive performance can be given a higher priority and lower time-slice and batch processes can have an increased time-slice with lower priority.

12. Interrupt Fencing

This technique is used to improve performance when there are a high number of interrupts on a server. Systems with high interrupts are often network and or disk intensive applications. On a busy system, you can see interrupts using the `mpstat(1m)`. CPUs taking a lot of interrupts are not as efficient at processing user threads since they spend a lot of time in the kernel processing interrupts. 

Interrupt fencing disallows user threads from being scheduled on CPUs that are receiving interrupts. Fencing allows interrupt and user threads to run more efficiently by reducing cache pollution and scheduling conflicts. There are a few different ways to create a fence for interrupts, but the easiest is to create a processor set on the CPUs that are taking the most interrupts.

To create a fence “in-place” without migrating interrupts, identify the active CPUs using `mpstat(1m)` and simply create a processor set on those CPUs. Once the processor set is created, simply don't schedule any user processes on this processor set.

When creating a “fence” for interrupts, the processor resource requirements must be taken into consideration. A balance

between interrupt processing and processors for user threads must be considered.

13. Database Design/Layout

Application performance can suffer from poor database design and layout choices. As application throughput is scaled, any point of serialization can become a bottleneck. The most common issue stems from improper choice of table and indexing structures.

Database vendors offer multiple choices for tables and indexes. Index only tables, hash tables, partitioned tables, clustered tables, reverse key indexes, partitioned indexes, bit-mapped indexes, and so on. Each of these structures effect performance depending on their usage.

Often, developers code applications and test applications on small machines. While these machines are fine for development, they can mask scalability issues. Oracle, for instance, has a default of one freelist per table. If tables are created and tested on a 1-4 processor machine, chances are no issues will not be seen. However, once these tables are placed on 72 processor server and there are 100's of threads inserting into a table with a single freelist, this can quickly become a bottle-neck. For another example of possible contention, consider standard b-tree vs reverse-key indexes.

Standard b-tree indexes sort by range. Consider the example where an index is created on column where a sequence number is incrementing automatically. As rows are inserted into this table, new inserts will be under the same leaf-node until the node fills up and splits. If the insert rate is high enough, contention on the block level will become unbearable. To alleviate this contention, reverse key indexes can be applied. Reverse-key indexes swap the

ordering digits of the number so subsequent inserts would not be necessarily in the same block. Reverse-key indexes work well for this situation, but they do not allow for range scans to occur – so there are trade offs that must be considered.

New in Oracle 10g is the Global Hash Partitioned Index. This index distributes sequential inserts via a hash function so as to avoid block contention.

14. DB configuration parameters

Database configuration parameters can help improve or limit performance. Buffer pool sizing, block size, memory page sizes, and latches are just some of the things which must be considered when configuring a database to scale. Most problems occur when applications are migrated from a small to a large machine.

Large machines have much more bandwidth, memory, and compute power. Memory access is non-uniform so there are optimizations which place processes into latency groups. Oracle 9iR2 and 10g create a DBWR per lgroup. Each DBWR process is responsible for cleaning LRUs associated with it's own lgroup. This reduces the overhead and improves the performance of buffer cleaning. Note that these NUMA optimizations are available starting with Solaris 9.

15. Application Scalability

Applications today are very complex so they must be designed to scale from the ground up. Careful attention to the data flow and threading are essential. For example, consider the Manugistics Fulfillment supply chain management application.

The Fulfillment application is based on two underlying technologies: the WebWORKS Foundation, a J2EE-based thin-client architecture, and their SWARM architecture, which distributes processing

nodes across available hardware resources. The combination of these features allows excellent vertical and horizontal scalability for Manugistics applications. You scale in multiple dimensions.¹¹

First off, the SWARM architecture divides the batch into multiple work units. Each work unit is taken from a queue and processed by a processing node. The processing node, simply a J2EE instance, can be run on either the local machine or a remote machine. Since work units are taken from a queue, they can be run on a variety of different machines running at various speeds. On large machines, multiple nodes are run on the same machine. This flexibility allows customers to configure systems that can be scaled to match their workload requirements.

16. Customer case study

Things are often not as they seem... OS data is not sufficient to diagnose application issues.

A customer submitted a service call about a period of slow performance on their new E25K. Every half-hour transaction throughput dipped to almost nothing for about 4 minutes. It took a while after an application restart for this to occur. The Sun service person was given a lot of Solaris performance data. After reviewing, there was no evidence of any problems – it simply looked like the system was not being driven by the application.

Further investigation turned up a script which was being run each half hour to clear the shared pool in Oracle. This was commonly done in the past as a work around to improve performance of SQL that was not shareable. It turned out that the throughput of this application on the new E25K was greater which in turn caused the pool to fill faster. A shared pool flush will

¹¹Sun Fire servers double supply chain management performance. SUPERG 2004

block all users until it has finished. If that wasn't bad enough, a shared pool flush is a single threaded operation.

This situation can be avoided by focusing on the application and writing SQL that can be re-used. Additionally, Oracle 8i introduced a parameter that cause Oracle to re-write statements at parse time "cursor_sharing".

Final thoughts

When scaling from a small server to a large server Memory, CPU, and IO must all be balanced. When going to large memory configurations with non-uniform access, OS changes that utilize MPO or large pages become necessary to maintain optimal performance. Solaris 9 introduced MPO changes and Solaris 10 has extended the scaling even further with regards to large memory optimization.

Growth in the number of processors puts additional stress on memory as well as the scheduling algorithms. Alternate scheduling classes and optimizations such as interrupt fencing are helpful to extend scaling.

The IO subsystem must be properly configured. Improper configuration with buffered IO can increase the memory pressure and cause additional over-head and fragmentation.

Understanding behavior is the key to building a scalable environment. Hopefully this paper has helped improve your understanding of how large servers are to be properly scaled.

Acknowledgments

I would like to thank the Sun SAE group, Sun Performance and Availability Engineering Group, Sun Market Development Engineering, and various Sun

customers for providing me insights into the unique use of this technology.

I would especially like to thank Bob Larson whose guidance continues to shape my work today. May you rest in peace.

References

Solaris 9 & 10 Reference Manual Collection [2002]. <http://docs.sun.com/>

Oracle Corporation. *Oracle 9i Database Performance Tuning Guide and Reference Release 2 (9.2) and 10g*. [2002, 2005]

"Avoiding Common performance when scaling Oracle 9iR2 applications" Fawcett [2003]. <http://www.sun.com/blueprints/>

"Performance Oriented Systems Administration" Larson [2002]. <http://www.sun.com/blueprints/>

"The Sun Fireplane System Interconnect" Charlesworth [2001]

"Solaris Internals" Mauro & McDougall, @2001 Sun Microsystems, ISBN 0-13-022496-0

"Sun Fire Servers double supply chain management performance" Fawcett, Kriekenbeck, & Dagistine. SUPerG [2004].

Lmbench and Stream information:
<http://www.bitmover.com/lmbench/>
<http://www.cs.virginia.edu/stream/>

Trademarks

Sun Fire is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries. Sun Fire sont des marques déposées ou enregistrées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the US and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. Toutes les marques SPARC, utilisées sous licence, sont des marques déposées ou enregistrées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

Solaris is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries. Solaris sont des marques déposées ou enregistrées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

Starfire is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries. Starfire sont des marques déposées ou enregistrées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.