# Making The Linux NFS Server Suck Faster

Greg Banks <*gnb@melbourne.sgi.com*>
File Serving Technologies,
Silicon Graphics, Inc.

**Abstract:** *The Linux kernel contains a great little NFS server, the emphasis being on `little'. As part of a strategy to sell Linux-based NAS servers as shared storage for compute clusters comprising hundreds or even thousands of nodes, SGI has modified the Linux NFS server to improve several aspects of scalability: many clients, high bandwidth, and high NFS call rate. Problems discovered during this work and SGI's solutions are discussed. All the kernel and nfs-utils changes described here are being made publicly available under the GPL licence, in stages beginning July 2006.*

## Introduction

Silicon Graphics, Inc. is widely known as a manufacturer of very large multiprocessor computers for the HPC (High Performance Computing) market; for example the Altix range's NUMA architecture scales to 1024 CPUs in a single system image. Less widely known is the fact that SGI is a vendor of storage hardware and software, including a NAS (Network Attached Storage) Server software and hardware bundle.

The fundamental principle of NAS is that the user's data resides on a central storage unit such as a RAID array, and is accessed remotely from client machines via well-defined network file sharing protocols. There are several protocols in common use today (e.g. CIFS, FTP, iSCSI[1]), but NFS and the Linux NFS server (knfsd[2]) will be the focus of this paper.

## NAS For Clusters

NFS is a popular means of accessing shared storage amongst users of compute clusters (such as HPC clusters or animation renderfarms). There are several reasons for this.

Firstly, NFS client software is typically free and comes bundled with the client operating system (for clusters, this almost always means Linux machines running 2.4 or 2.6 series kernels). Large scale NFS deployments using automount and NIS maps are well-understood solutions.

Secondly, NFS can be run on a cheap commodity protocol stack, right down to the wiring. Solutions using FibreChannel or Infiniband may exhibit lower latency or higher throughput, but for many cluster users 1 Gigabit Ethernet is adequate.

Thirdly, NFS does not place any arbitrary limit on the number of participating nodes, as some cluster filesystems do. Implementation limits and resource limits apply of course, but an NFS user has every reason to expect that by buying a large enough server they can serve as many clients as is necessary.

Finally, NFS is simple[3]. The basic protocol is reasonably well defined in freely available public documents (RFCs), and does not suffer from a surfeit of complexity or incompatibilities between slightly different versions.

The price paid for this simplicity is a poor approximation of POSIX filesystem semantics. For example, using the *mmap* system call on NFS files can sometimes give unexpected results. However, many cluster users run software whose demands on the filesystem are for performance rather than strict semantics. In particular, and perhaps surprisingly given that files are being shared between hundreds of machines, the weak data consistency model used

1 iSCSI is block-based and thus not strictly a filesharing protocol, but is commonly provided on NAS products.

2 The `k' in knfsd is because the server runs inside the Linux kernel. There is also a unfsd, now little used.

3 NFSv4 less so.

by NFS usually does not seem to be an issue.

An advantage of NFS' simplicity is the relative lack of emergent behaviours[4]. Proper NFS clients are designed to handle servers being rebooted or being removed from the network with a minimum of disruption, and in practice this goal is often achieved. By contrast, the complex interactions and inter-dependencies between nodes characteristic of some cluster filesystems can lead to instability.

From SGI's experience, a picture of a "typical" cluster, from the point of view of the shared storage server, emerges:

- Nodes are Linux 2.6 or 2.4 clients, running either a RedHat or SUSE Enterprise distro, or a cheaper alternative such as CentOS or Fedora Core.
- Cluster users have Linux expertise in-house, and can roll out custom Linux kernels to fix client bugs. This is important for a server vendor as it gives some flexibility in preparing a solution.
- Nodes are small machines, for example 1 or 2 CPUs.
- There are many nodes, maybe two thousand or more.
- Connectivity is usually Gigabit Ethernet. Network infrastructure is sophisticated, with a hierarchy of very carefully managed switches, some of them very large.
- The throughput/IOPS requirements on each cluster node are relatively low.
- Aggregate throughput/IOPS requirements on the server are high. When throughput is the main requirement, often one gigabyte per second is required. When IOPS are the main requirement, often 50 to 100 fast (15 KRPM) disks are required.
- To achieve the required performance, the server needs multiple Gigabit Ethernet cards. For example, 1 gigabyte per second would require 8 cards if all the cards could be run at line rate.
- The name of the server as used by cluster nodes needs to be uniform across the cluster, to ease administration of the cluster. This can mean either a single IP address using Ethernet link aggregation or a single DNS hostname using round-robin DNS[5].
- A global file namespace is needed; instances of the application running on any node in the cluster need to see the same files on the server.
- I/O patterns vary between users, but are highly regular and predictable for a given user and application. Unfortunately, cluster users like to divide their clusters into multiple partitions and run different jobs with different I/O characteristics in those different partitions.

## SGI's Approach

The design that SGI sells to meet cluster users' needs for shared storage is based on small numbers of large individual servers. For many users, a single Altix server with 8 CPUs and 8 Ethernet cards and a single RAID array will meet the performance requirements.

This approach has several advantages. For example, a single server trivially achieves a global namespace.

Another advantage is caching. Some workloads benefit from the caching of data and metadata; the large, high-bandwidth shared memory of the Altix architecture allows caching to be maximised.

The building block for NAS servers is the Altix A350 "brick", which is an ia64 NUMA node containing 2 CPUs, 12 DIMM slots, and 4 PCI-X slots. These can be chained together into a single machine using SGI's NUMAlink cache-coherent interconnect. Thus, an 8 CPU Altix[6] can be fitted with up to 192GB of RAM using 4GB DIMMs.

The design rules seek to balance CPU, memory, network I/O and disk I/O throughout each NUMA node, minimising NUMAlink traffic between nodes. The goal is to have each NUMA node acting as much as possible like a separate server, while retaining the advantages of shared data and metadata caching.

This design pre-dates SGI's move to Linux for

4 Again, NFSv4 less so.

5 Aggregation and load-balancing options are limited by the network topology.

6 This is quite small by Altix standards.

storage servers. It is based on characteristics of the filesystem, the kernel NFS server in the Irix operating system, and the MIPS-based Origin hardware used in previous generations of the design. For this design to succeed, the Linux kernel NFS server needs to scale to handle large amounts of NFS traffic on multi-node NUMA machines.

## Performance Problems

Unfortunately, early trials of the Altix NAS Server showed that the Linux NFS server in stock SUSE Linux Enterprise 9 [SLES9] did not perform well nor did it scale that performance.

The first tests performed were streaming reads from page cache, using TCP and a 32KB block size.

At the low end, a single A350 brick with 2 clients was unable to maintain line rate on 2 Gigabit Ethernet NICs (being CPU limited).

Adding more hardware did not help. For example, adding a fourth brick (going from 6 to 8 CPUs) and 2 more clients, added 33% more CPU power but achieved only 12% more NFS throughput.

Another interesting phenomenon was that with a fixed load of 4 clients, going from 4 CPUs to 6 CPUs used up all the extra CPU time but actually delivered less throughput in absolute terms. In other words, negative scaling.

Furthermore, with workloads which emphasize NFS call rate, the server quickly entered a "locked-up" state where all the CPU was spent servicing NFS requests and userspace tasks did not run for minutes at a time. As the first two steps in the NFS mount process involve userspace daemons, this resulted in the server becoming unavailable to new clients.

This paper describes the work needed to solve these problems.

## Principles Of Operation

The Linux NFS server comprises two main userspace daemons and a number of kernel threads. The userspace daemons (portmap and rpc.mountd) are mostly used during the mount process, the kernel threads do most of the work.

Portmap is the RPC rendezvous daemon; it manages an in-memory database of mappings between RPC program numbers and TCP or UDP ports. Querying this database is a necessary first step in setting up any RPC communication, including to rpc.mountd.

Rpc.mountd has several functions, but most significantly it is called by clients at mount time to return the root file handle for an exported directory. Thus it interprets data in the */etc/exports* file and enforces the permission policies contained there.

The kernel threads are visible in the process list as multiple "nfsd" threads and one "lockd" thread. The nfsd threads serve the main NFS protocol and the lockd thread serves the auxiliary NLM protocol which implements file locking over NFS. All the nfsd threads are effectively in a global pool of threads, and they handle individual NFS calls from the network on a call-by-call basis. Note that there is no relationship between threads and clients; indeed there is very little per-client state in the server[7].

The kernel code layer which deals with reading incoming RPC calls from the network layer and managing nfsd or lockd threads, is the sunrpc layer, located in the *net/sunrpc/* and *include/linux/sunrpc/* directories. Most of the work described in this paper comprised modifications to that layer.

The main sunrpc data structures are

- *svc_rqst* which holds per-thread state.
- *svc_sock* holds per-socket RPC state.
- *svc_serv* holds state for a service (such as NFS), i.e. is effectively global. It contains a list of sockets which have data waiting to be read, a list of idle threads, and lists of permanent (e.g. TCP rendezvous) and temporary (e.g. connected TCP) sockets.
- *svc_export*[8] represents an export point from the */etc/exports* file. A collection of these is maintained in the kernel, initially empty and populated on demand by rpc.mountd during the mount process.

The main communication mechanism between

7 Again, NFSv4 differs.

8 Despite the name, this is an NFS data structure, not a sunrpc one.

the kernel NFS server and the userspace daemons is the nfsdfs filesystem, a small special-purpose filesystem which is usually mounted on */proc/fs/nfsd*. It contains a small number of files with interesting semantics (somewhat like the */proc* filesystem itself). For example, the *threads* file can be read to get the number of nfsd threads, and written to change the number of threads. Other files are used to implement "upcalls" to rpc.mountd when the kernel needs a policy decision to be made.

The lifetime of an RPC service thread is:

- if no socket has pending data, block (this is the normal idle condition)
- take a pending socket from the global list
- receive an RPC call from the socket
- decode the call (callout into protocol-specific code)
- dispatch the call (another callout; this is where I/O the exported filesystem occurs, e.g. reading a directory)
- encode the reply (another callout)
- send the reply on the socket.

Note that the thread has exclusive access to the socket while receiving and sending, but not when dispatching. This allows an nfsd thread to block waiting for filesystem I/O without preventing other threads from handling calls from the same client.

## What Is Scaling?

The specific goals of the scaling work were to make the Linux NFS server scale NFS throughput and NFS call rate as close to linearly as possible and as close as possible to hardware limits, from the smallest SGI NAS server configuration (2 CPUs and 2 Gigabit Ethernet NICs) to the largest (8 CPUs and 8 NICs). Specific workloads which must be handled are: a small number of read streams, a small number of write streams, a large number of read streams, a large number of write streams, a readonly directory-traversal, and the SPECsfs benchmark [SPE97].

These workloads must still perform well when the load is generated by 200 or more clients instead of just a smaller number. Note that scaling to many clients is a different and additional problem to scaling for high performance, see [KEG06] for similar work for userspace network servers.

## Lock Contention & Hotspots

The major performance factor was a series of global locks which were contended. The general procedure for diagnosing these is kernel profiling, using either Oprofile [LEV] or SGI's *profile.pl* tool.

Note that in a NUMA architecture, we can experience an effect which has much of the same performance impact as lock contention, but without the actual contention (which can be diagnosed in various well-known ways such as lock metering or kernel profiling [LEV]). All that is required is a cacheline which is frequently written by CPUs from multiple nodes. The additional latency involved in the cache coherency transaction which recalls a dirty cacheline from another node (while the CPU is stalled) is enough to dramatically increase CPU usage. This extra time appears in kernel profiles but without any apparent connection to lock traffic.

The most important culprit was the *svc_serv* spinlock, which guards the global lists of pending sockets and idle threads. The cacheline containing the lock and the lists is written by every thread at least twice for each call the thread handles.

The solution was to split those fields out of the *svc_serv* structure into an array of new *svc_pool* structures, and allocate one such pool for each NUMA node. Each *svc_pool* is assigned a subset of the nfsd threads, and those threads are constrained to only run on CPUs of one node by setting each thread's mask of allowable CPUs when it is spawned. Sockets are temporarily assigned to *svc_pool*s only briefly while calls are being decoded. These choices ensure that cachelines do not travel off-node, at least for the common case on every call.

Part of this solution was to move the code which handled the detection and disconnection of idle temporary (i.e. connected TCP) sockets, from the main call path to a timer which runs every few minutes. Previously the aging

algorithm depended on maintaining a global LRU list of temporary sockets. Replacing this with a mark-and-sweep algorithm allowed the main call path to be shorter and simpler, to hold the lock for shorter times, and to avoid a list manipulation which will dirty additional cachelines.

Another hotspot is the *nfsdstats* structure, which contains various global NFS statistics counters. The cachelines of this structure are written several times per call, but the most egregious of these instances is the nfsd thread statistics. Twice per call, each nfsd thread acquires a global spinlock (another contention point) before updating a set of counters. Those counters are meant to provide a sysadmin with enough information to be able to manually tune the number of nfsd threads; in practice they have proven difficult for real sysadmins to use. The short term solution was to remove those statistics; the long term solution would be to replace them with an automatic control loop.

For read workloads, the global spinlock which protects the readahead parameters cache hash index is a contention point; the solution was to split the lock into multiple locks indexed by the low bits of the hash value.

Another contention point is the spinlock which protects the *ip_map* cache hash table. This hash table is looked up on every call; with many client IP addresses and the *ip_map* hashing bug mentioned below (see **Mountstorm**) this spinlock can use a significant amount of CPU time. Because the lookup returns the same object almost every time for a given client, the solution was to avoid the lookup entirely by caching that pointer in the *svc_sock*.

## NUMA Factors

A major performance factor, at least on Altix systems, was the tendency of the CPU scheduler to provide poor locality of reference for nfsd threads.

Individual threads would jump from CPU to CPU without any apparent regard for whether the per-thread data would be present in CPU caches.

Threads would send NFS replies (which usually involves multiple trips all the way down to the Ethernet driver) without any regard for the I/O access latency between the CPU doing the sending and the NIC. On a NUMA machine, this is an important factor which affects the amount of CPU time spent waiting for PCI I/O to complete. The ideal situation is always to send replies from a CPU on a NUMA node nearest to where the NIC is located.

The splitting of *svc_serv* into multiple *svc_pool*s described above is also the solution for these problems. To reduce contention on the *svc_pool* locks, the nfsd threads are made node-specific by setting the mask of allowable CPUs for each thread when it's spawned. If the NICs are configured so that their interrupts are bound to a single local CPU, the combination of the two restrictions ensures that sending NFS replies always occurs from a CPU close to the NIC.

Note that this requires a fixed binding of NIC interrupts to CPUs. This is always the case on Altix platforms, but on i386 and x86_64 platforms the *irqbalanced* daemon needs to be disabled and interrupts manually bound to CPUs. In SGI's experience, *irqbalanced* is too eager to move interrupts between CPUs, harming locality and generally resulting in poorer performance; disabling it has no known ill effects.

## Mountstorm

One characteristic of cluster job management systems is that they tend to result in extremely bursty NFS mount traffic. For example, when a new job starts on 200 nodes the server will see 200 mount requests within a second or two. For the cluster job to proceed, all of these near-simultaneous mount requests must complete, successfully, within the short period of time determined by the clients' TCP connection timeout and RPC call timeout. We call this condition a *mountstorm*.

To handle mountstorms, we want to take advantage of the server's multiple CPUs and process as many as possible of those mount requests in parallel. Unfortunately, Linux serialises mount requests in three places.

Firstly, the portmap daemon is single threaded.

Secondly, the rpc.mountd daemon is also single threaded. Normally that thread needs to perform a blocking DNS reverse lookup and a blocking DNS forward lookup for each connecting client. A slow DNS server can be the limiting factor in a mountstorm. One workaround is to enter all the clients' IP addresses in the server's */etc/hosts* file, to ensure the hostname lookups are local. Another workaround (not tried) would be to enable hostname caching in the Linux name service caching daemon.

Thirdly, we have the problem of the *ip_map* cache in the sunrpc code. This is a data structure, indexed by a hash table, used to map client IP addresses to authentication information. At mount time, the first incoming NFS call from the new client fails to find a matching entry in this data structure, and an upcall to rpc.mountd is used to fill it in.

On older kernels (such as SLES9), a bug in the sunrpc IP address hashing code on little-endian 64bit machines (such as Altix) results in nearly all the *ip_map* entries being hashed to the same hash bucket, thus effectively reducing the hash table to a linear list [CAP05]. The hash table is protected by a global spinlock, so searching is single threaded. When 2000 clients try to mount the server at the same time, the machine appears to lock up, with a single CPU walking this enormous list and every other CPU spinning on the global lock, and no spare CPU for rpc.mountd to run.

The solution was to backport a simple patch to fix the *ip_map* hashing function from the Linux mainline. When combined with the */etc/hosts* workaround, the SGI NAS server has demonstrated the ability to handle a mountstorm of 2000 clients.

After the first generation of SGI's Linux-based NAS solution shipped, discussions [BRO06] on the Linux NFS mailing list showed that the second problem – the single threaded nature of rpc.mountd – could be solved correctly and relatively simply by making rpc.mountd multi-threaded. The original expectation was that this would be a difficult exercise, because writing multi-threaded RPC servers is normally quite difficult. However, it turned out that merely forking multiple rpc.mountd processes at the correct point in the *main* routine (after registering the service socket and before entering the main RPC loop) was sufficient.

This was because the rpc.mountd code keeps almost all of it's state either in the kernel, or in files which are read-mostly and already needed to be file locked when accessed (to avoid racing with programs like *exportfs*). An nfs-utils patch has been submitted and should appear in a future release of the SGI NAS server, entirely removing the need for the */etc/hosts* workaround.

## Duplicate Request Cache

The Linux NFS server contains an implementation of the duplicate request cache described in [CAL95], [CAL99] and [KIR06], and known in Linux as the repcache[9]. Briefly, this is a data structure which contains a binary copy of the reply sent for every non-idempotent NFS call received by the server (non-idempotent calls are those which cannot be safely repeated, like RENAME). If, for some reason, a reply is sent by the server and not received by the client, the client will retry the call. The repcache allows the server to resend the original reply without trying to re-run the call, which would result in a failure or other unexpected behaviour.

Normal operation of the repcache implies that every non-idempotent call involves a search of the repcache before dispatch, and an insert into the repcache after dispatch. The repcache is indexed by a small hash table, but that hash table is protected by a single global spinlock. Testing using a synthetic workload comprising exclusively non-idempotent calls showed that this spinlock and the hash table are global hotspots.

Another problem with the repcache is that the maximum number of entries is fixed at 1024, a number that has not increased since it was chosen in the mid 1990s. Consider that an entry needs to remain in the repcache for at least one client timeout period, which is typically at least 1.1 seconds (sometimes longer). Thus with a fixed size repcache of 1024 entries, any

9 Other implementations call this structure the dupcache.

workload which contains more than about 1024/1.1 = 930 non-idempotent calls per second will flood the repcache and render it ineffective.

Note that the WRITE call is non-idempotent, and it's not unusual for parts of a cluster workload to have multiple thousands of WRITE calls per second. Furthermore, the expected maximum call rate for the SGI NAS server would be nearly $10^5$ calls/sec, two orders of magnitude higher than the maximum rate at which the repcache is effective.

It has been proposed (for NFSv4 [EIS05]) that NFS clients should not emit retries at all when used on a reliable transport such as TCP. Furthermore, TCP's transport-level retry behaviour is known to reduce the incidence of duplicate calls. So arguably, one possible solution to the repcache's problems would be to disable both UDP support and the repcache, or (less drastically) to bypass the repcache for calls over TCP.

This author's opinion is that as long as there exist NFSv3 clients which retry over TCP or use UDP, the repcache will be necessary. Client retry times can be tuned downwards, and any number of effects (e.g. overload conditions) can cause service times to rise to significant fractions of a second.

The solution chosen was to completely modernise the repcache. The first major step was automatic expansion of the repcache under non-idempotent load, triggered by the largest age of a record falling below a fixed threshold. The problem of shrinking this data structure was not addressed, as testing showed that at the maximum expected call rate the memory usage was only 16 MiB.

A further enhancement was expanding not just the repcache but the hash table index too, using a progressive lazy rehashing technique to spread the resizing overhead over time. However, calculation and testing showed that (perhaps surprisingly) the dynamic range of repcache sizes was small enough that a single fixed hash table size could be chosen which would result in acceptable hash chain lengths for all loads.

The second step was splitting the repcache's hash table spinlock into multiple spinlocks, one per hash table bucket, in order to reduce contention. As that spinlock also protected other data structures, most notably the global LRU list, those structures were also split into one per repcache hash bucket.

Finally, the repcache hash algorithm was tweaked so that the set of bits influenced by the client IP address was distinct from the set of bits influenced by the transmission id (XID). With NFS server NUMA-awareness, calls from a given client are usually only ever handled by CPUs in a single NUMA node; so this technique reduces contention and hotspotting of repcache hash table buckets.

## CPU Scheduler Overload

On stock SLES9 configured with many nfsd threads, workloads which feature high NFS call rates from many clients hit an unexpected problem. Each incoming call causes an nfsd thread to be woken; soon almost every nfsd is runnable but only a few can actually run on a CPU. This results in very long CPU scheduler queues. For example, with 128 threads and 4 CPUs, a load average of over 120 has been measured.

These long runnable queues result in the SLES9 CPU scheduler using up the last few percent of available CPU time. Also, nfsd threads are scheduled preferentially to userspace tasks. The system eventually enters a state where nfsd threads run normally but userspace programs are not scheduled for minutes at a time.

In this state, existing NFS clients still have their calls serviced, but because portmap and rpc.mountd cannot complete TCP connection setup by calling *accept*, no new clients can mount the server.

The solution was a simple tweak to limit the number of nfsd threads woken but not yet running. The sunrpc code tracks how many threads are in this state and does not attempt to wake new threads if that number exceeds a small threshold, currently 5 per *svc_pool*. The result is a load average under load of 5 to 6 per NUMA node, and a complete absence of the overload condition.

## NFS Over UDP

Testing streaming read and streaming write workloads showed that NFS over UDP is limited to about 145 MiB/s, or the equivalent of about one and a half Gigabit Ethernet link's worth of throughput, no matter how many cards are used. Discussions and experiment showed that the limit is due to the NFS server using only a single global UDP socket for all NFS/UDP traffic. By contrast, NFS/TCP uses a single socket for each client connection.

An nfsd thread needs exclusive access to the socket while receiving and sending data. In particular when sending the thread holds the *svc_sock* lock while using the global UDP socket as an *sk_buff* queue for constructing a datagram. So only one thread at a time can run the entirety of the network stack send path: waiting for socket send queue space, routing (done for every UDP datagram), IP fragmentation, and thirty-odd calls to the driver's *hard_start_xmit* routine.

During the course of this work, a solution was coded and tested for this problem. Ultimately, the solution could not be shipped in the SGI NAS server because it required a change in code which cannot be upgraded with a replacement kernel module.

The solution is to use multiple UDP sockets. Linux already has an option to bind a socket to a network device. This allows multiple sockets using the same UDP port to co-exist and function normally, as long as they are bound to different devices (plus one socket which is not bound to any device). When a UDP datagram arrives on a card, the network stack tries to choose a UDP socket bound to that card before falling back to the unbound socket. The network stack also provides a notifier callback chain which the NFS server can use to discover what devices exist, even if they are created after the NFS server initialises. The solution uses these two existing features, and creates a device-bound UDP socket on the NFS port for every non-loopback network device.

Unfortunately, device binding affects the network stack's outbound routing decision[10], so using these same sockets to send NFS replies will fail if the server uses asymmetrical routing. The solution is to use a different set of UDP sockets for sending, one per CPU. A socket option, UDP_SENDONLY, was added; this option prevents a socket from being entered into the UDP socket hash table which is used when receiving UDP datagrams. As a result the socket can be used to send datagrams (without any effect on the routing decision) but not receive them.

Testing showed that this solution provides almost the same throughput as a TCP-based solution, breaking the 145 MiB/s limit.

However, given the multitude of other problems with using UDP as a transport protocol for NFS, this is not necessarily a good thing. SGI continues to recommend that customers use TCP where possible.

## Write Performance To XFS

When all these NFS changes were being tested for performance, one workload remained too slow: streaming writes to a small number of files on disk. After much testing this was eventually tracked down to a bug in the XFS code which affected overall performance only when the filesystem was written to via NFS, not for local I/O.

The XFS writepages code was incorrectly interpreting the flag passed to it from the VM which indicated whether it could block. So, on write congestion kupdated would block in XFS, while holding the `i_sem` lock on one of the files being written. This prevents nfsd (when handling the WRITE RPC) from taking `i_sem` to add new dirty pages to the page cache. As a result, the server either reads from the network, or writes to the disk, but cannot do both at the same time, which effectively halves NFS write performance. The XFS team have solved this problem.

## Tunings

Several kernel tunables were found to have beneficial effects for NFS serving.

- Maximum TCP socket buffer sizes (the 3rd

10 As pointed out by Neil Brown.

number of *net.ipv4.tcp_rmem*) were increased to 512 KiB, so a large window scale is chosen during TCP connection.

- Interrupt coalescing parameters (*ethtool -C rx-usecs rx-frames*) for the tg3 Gigabit Ethernet driver were changed to cause an interrupt about every 20 Ethernet frames (on older kernels, the default was to interrupt the CPU for every frame). This reduces softirq CPU usage at high packet rates.
- The parameters *vm.dirty_background_ratio* and *vm.dirty_writeback_centisecs* were decreased to 10% and 50 cs, so that the VM will begin writing dirty pages to disk earlier. The goal is to improve streaming write performance by reducing the time taken to service COMMIT calls (because some or all of the unstable data has already been written to disk).
- The *async* export option was explored, in conjunction with the above VM parameter changes, in an attempt to improve streaming write performance. It does improve performance, but unfortunately at the cost of increasing the chances of out-of-memory deadlock if the client has more RAM than the server. Furthermore, it breaks documented NFS semantics and is a data loss risk.
- The *no_subtree_check* export option was used. This option is documented as having a security impact (allowing a skilled attacker to synthesise filehandles for files which they should not be able to access). However, the security issue does not apply if all export points are the roots of filesystems, a constraint which happened to be true for SGI NAS systems. Conversely, on workloads which use very large numbers of files and directories, such as the standard SPECsfs benchmark, using *no_subtree_check* can save up to 10% CPU.
- The ARP code was configured only to respond to ARP requests from the NIC on which the request was received (*net.ipv4.conf.ethN.arp_ignore=1* and *...arp_announce=2*). This avoids the problem seen on shared media network configurations, of streams of packets from clients jumping randomly between NICs, destroying the NUMA locality of reference.
- The maximum ARP cache size was increased to 2048 to prevent the table overflowing at 1024 unique MAC addresses.

## Performance Results

The results from two tests give some idea of the performance improvements which result from all the above changes. The Before case is the stock SLES9 SP2 kernel (plus some tg3 driver patches which appeared in SP3); the After case is the same kernel with all the above patches applied. Both kernels were tuned as described above.

The first test is a multiple streaming reads from a separate file per client, from the server's page cache. The transport protocol was TCP and the block size was 32KiB (the maximum block size supported by the stock SLES9 kernel). The configurations tested used matched numbers of CPUs, tg3 Gigabit Ethernet NICs, and Origin O350 test clients, and were achieved by progressively connecting A350 bricks with NUMAlink. The server and clients were connected via a Foundry FastIron switch; Ethernet bonding was not used. Data was collected using SGI's Performance Co-Pilot [PCP]. Two metrics are shown: aggregate throughput in MiB/s

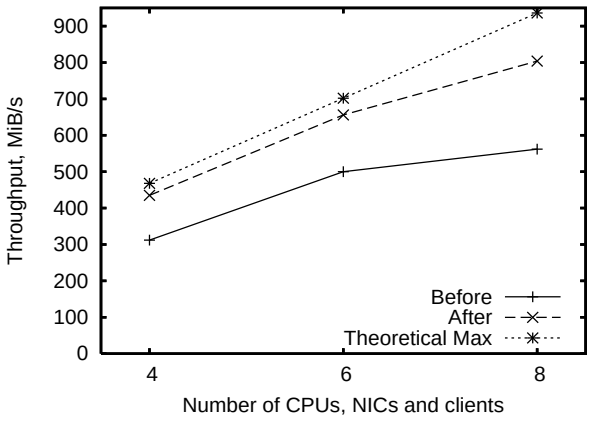


*Fig 1: Streaming read throughput*

and CPU usage (system plus interrupt)

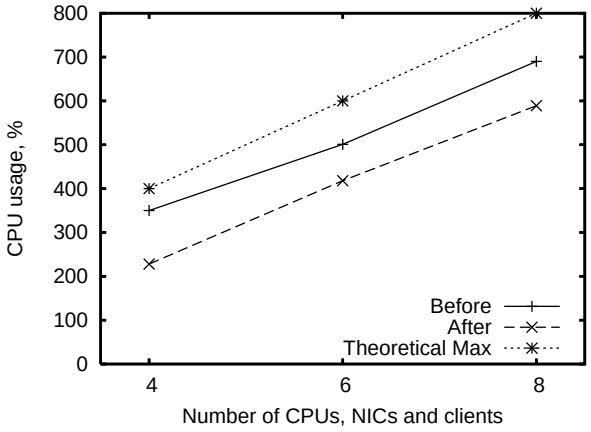


*Fig 2: Streaming read CPU usage*

The most obvious feature is that NUMA-awareness and reduction of global contention points have significantly increased the throughput and reduced the CPU usage for all the tested configurations. In particular, throughput is now much closer to the theoretical maximum. But note that throughput scaling is still not linear; work remains to be done in this area.

The second test uses a synthetic load generating program to simulate varying numbers of clients each performing a recursive directory traversal over a tree small enough to fit into memory[11]. The simulated clients reproduce the pattern of NFS calls recorded for the *rsync* program running on a Linux 2.4 client (although this pattern is basically the same for the *ls -lR* or *find* programs).

The 2.4 client had poor attribute caching and did not use READDIRPLUS, so this workload comprises mostly redundant ACCESS and GETATTR calls. Thus, one of the features the test measures is the NFS server's ability to handle high call rates.

The test program uses multiple threads on each client machine. Each thread has its own IP address and its own TCP socket. Thus the test also measures the effect on the NFS server of varying numbers of connected TCP sockets and client IP addresses (but not MAC addresses, unfortunately). The threads do not implement rate control, but issues NFS calls as fast as the server can respond. A real NFS client would do some additional processing between calls, so each virtual client provides a load equivalent to an estimated 2 to 4 real clients.

The server was a fixed configuration of 2 A350 bricks, with 4 CPUs and 4 tg3 Gigabit Ethernet NICs. The server and clients were connected via a Foundry FastIron switch; Ethernet bonding was not used. Four Origin O350 client machines were used, and the number of virtual clients varied.

Two metrics are shown: NFS calls per second (IOPS)

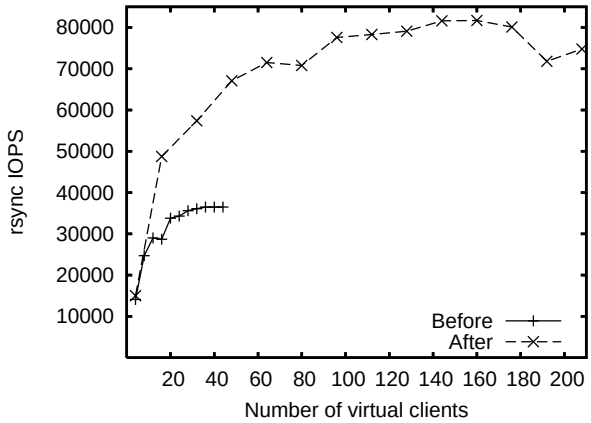


*Fig 3: IOPS under rsync workload*

and CPU usage (system plus interrupt).

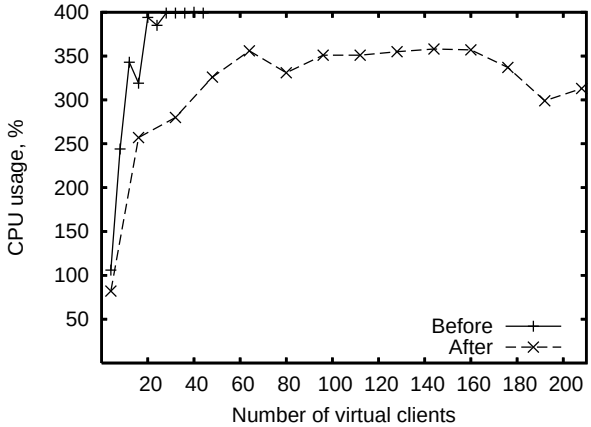


*Fig 4: CPU usage under rsync workload*

The first visible feature of the graphs is that the saturation call rate for the After case is about twice that of the Before case. This is partly due to NUMA-awareness and reduction of global contention points, and partly due to reduction of per-call CPU overhead. This same effect also

11 A copy of an i386 RedHat9 install tree, approximately 119000 inodes.

resulted in a near doubling of SPECsfs benchmark performance, and enabled SGI to publish SPECsfs results for the Altix platform for the first time.

The second visible feature is the CPU overload phenomenon described earlier. After 44 virtual clients, the Before kernel uses all of last few percent of CPU. New client threads cannot mount because portmap and rpc.mountd are not scheduled often enough; there is no data because the userspace statistics collection daemon is not scheduled often enough. In contrast, the After kernel handled the load well and showed no signs of failure when the test was terminated at 208 virtual clients.

To illustrate the difference these patches made, we can consider two competitive NFS server evaluations conducted by customers. In July 2005, customer “W” tested a system very similar to the Before kernel with 200 clients and the Linux NFS server failed to achieve either throughput or IOPS requirements in several tests. In May 2006, customer “P” tested the After kernel with 2000 clients and the Linux NFS server passed with flying colours.

## Future Work

Version 3 of the NFS protocol does not have any provision for the client to notify the server of the lifetime of client-side file descriptors. This means the server cannot keep actual file descriptors open for the files it exports to clients, and must perform the equivalent of opening and closing a file descriptor on every RPC from a client. For a streaming write workload, this results in XFS being forced on every single WRITE call to truncate the speculative allocations it makes when it detects streaming writes.

This has two negative effects. Firstly, traffic to the XFS log is greatly increased, and for some workloads and filesystem layouts this might be the performance limiting factor. Secondly, with multiple streaming writes under some conditions, XFS' smart extent allocation may be completely defeated, resulting in on-disk fragmentation and reduced performance.

One obvious approach to solving this problem is to change the NFS server's existing cache of read-ahead parameters to be an open-file cache. Another alternative would be to extend the open file data structure used for NFSv4.

For NFS writes, all incoming data is subjected to two memory-to-memory copies. The data is first copied from the network stack's sk_buff data structures into the nfsd thread's buffer pages, then from there into the page cache. The first of these copies is unnecessary, and could be eliminated by changing the interface between the sunrpc code and the network stack. A useful side effect would be a large reduction in the memory usage of each nfsd thread, as most of the thread's buffer space would no longer be needed.

To help deal with the mountstorm problem, the portmap daemon could be made multi-threaded. This will not be quite so simple as rpc.mountd because portmap has an in-memory database and so does not currently need to do any locking. Also, any significant changes to portmap run the risk of clashing with the IPv6 port being done by Groupe Bull.

In addition, rpc.mountd and the kernel sunrpc code could be modified to prime the *ip_map* cache when handling the MOUNT call, thus removing the need for the upcall to rpc.mountd on the first FSINFO call. This would eliminate a trip through rpc.mountd from the mount process, and avoid the burst of *ip_map* cache misses (and any contention on the global *ip_map* cache spinlock) during a mountstorm.

There are still some global hotspots which could be fixed, in particular the *nfsdstats* structure.

The repcache could be made to shrink under memory pressure. The repcache hash table index could be made smaller by default and allowed to expand under load, like the repcache itself.

## Acknowledgements

This talk describes work performed at SGI Melbourne, from July 2005 onwards. Kernel & nfs-utils patches described are being submitted from July 2006 onwards.

Thanks to code reviewers Neil Brown, Andrew

Morton, Trond Myklebust, Chuck Lever, Christoph Hellwig and others.

Thanks to those who reviewed drafts of this paper, including James Peach, James Yarbrough, Peter Leckie, Mitch Davis, Mark Goodwin.

## References

[BRO06] Neil Brown et al, *2.4 vs 2.6*, linux-nfs mailing list, sourceforge.net, May 2006.

[CAL95] Brent Callaghan, Brian Pawlowski, Peter Staubach, *RFC 1813 NFS Version 3 Protocol Specification,* Internet Engineering Task Force, Jun 1995.

[CAL99] Brent Callaghan, *NFS Illustrated*, Addison Wesley, ISBN 0201325705, 1999.

[CAP05], Don Capps, *An interesting performance thing?*, linux-nfs mailing list, sourceforge.net, 14 Dec 2005.

[EIS05], Mike Eisler, *Retries in NFSv4 Considered Harmful*, *http://nfsworld.blogspot.com/*, 19 Apr 2005.

[KEG06] Dan Kegel, *The C10K Problem*, *http://www.kegel.com/c10k.html*.

[KIR06] Olaf Kirch, *Why NFS Sucks*, Proceedings of the 2006 Linux Symposium, Aug 2006.

[LEV] John Levon, Phillipe Elie, at al, Oprofile profiling tool for Linux, *http://oprofile.sourceforge.net/*

[PCP] Silicon Graphics Inc., Performance Co-Pilot, *http://oss.sgi.com/projects/pcp/*.

[SGI] Silicon Graphics Inc., Storage Page, *http://www.sgi.com/storage/*

[SPE97] Standard Performance Evaluation Corp., SPECsfs benchmark, *http://www.spec.org/sfs97r1/*