Title: Linux Kernel ZFS Port for Lustre Servers

Author: Brian Behlendorf <behlendorf1@llnl.gov>

Date: May 23rd, 2008

Summary: For petascale Lustre deployments NNSA and CEA/DAM recognize the need for a strong underlying file system for Lustre Object Storage Targets (OST). One candidate is Sun's ZFS which contains many of the needed features and is available under the CDDL open source license. To properly evaluate ZFS LLNL has ported the DMU component of ZFS to the Linux kernel for analysis.

Approach: The strategy taken to port the DMU to the Linux kernel was to make as few modifications as possible to the DMU code proper. This allows us to easily re-base the port on the latest Sun ZFS source code which is under active development. To achieve this a shim layer was written, the Solaris Porting Layer (SPL), which provides a Solaris style API the DMU code can call directly. Internally the SPL maps this API to equivalent Linux kernel primitives. In the few cases where this approach does not work patches are applied directly to the DMU source.

Services Provides by the SPL Layer
Adaptive Mutex Interface
Memory /Slab Allocation Interfaces
Kernel Thread Interface
Read/Writer Lock Interface
Condition Variable Interface
Task Queue Interface
Block/Character Device Interfaces
Timer/Clock/Delay Interfaces
Panic/Assertion/Debug Interfaces
Kstat Interface
Credential/Callback/RNG/Misc Interfaces
SPL Test Infrastructure and Regression Suite

To characterize the performance of our kernel DMU port we then ported Sun's PIOS benchmark in to the Linux kernel. Renamed KPIOS this code can be used to simulate a large random lustre IO load in a single node test environment greatly simplifying the needed test infrastructure.

Results: The results of this investigation were encouraging. KPIOS can currently achieve IO rates as high as 79% of the expected peak write rate and 95% of the expected peak read rate under a 1MiB random IO workload. This expected peak rate is defined to be the observed sustained aggregate parallel streaming IO rate to all attached disks. The following modifications to the DMU were made during our investigation and they are listed in descending order of impact. A summary of these performance results can be found in Appendix A and a detailed analysis is available in the attached spreadsheet.

- **Disable Prefetch**: For reasons which are still not entirely clear disabling the DMU level prefetch dramatically improved read performance. A detailed analysis must still be done to clearly understand the complicated interplay between the DMU and Linux IO elevators.
- **Zero copy**: Copying data to/from an internal file system buffer before reading/writing it from disk is a well known performance bottleneck. While zero copy interfaces do not yet exist for the DMU we simulated this behavior in our testing to determine the expected performance improvement.
- **vdev_max_pending**: This is a DMU tuning which controls how many concurrent IO requests can be issued to the underlying block devices. Increasing this value greatly improved performance in the read case by allowing the Linux IO elevator to merge the relatively small IO requests (128k) issued by the DMU in to much larger requests before being issued to the disk. The penalty for this is increased latency and we are slightly subverting the DMU elevator which has a much better notion of IO priority. Ideally, we need to get the DMU to be able to submit large IO requests to the Linux elevator which can then use the 'noop' scheduler.
- **zfs_arc_max**: Carefully tuning the DMU ARC size which controls how much data is cached is critical in the current implementation. When the ARC is tuned too small the DMU will be unable to keep the drives busy and performance will suffer. Conversely if you allow the ARC to grow too large you hit performance issues due to the DMUs need for large expensive contiguous memory allocations. This value can be tuned effectively for benchmarking, but in the long term the DMUs appetite for large contiguous allocations must be quelled likely with a scatter/gather implementation.
- **Disable Checksums**: One major advantage of ZFS is that all data is checksummed and thus silent data corruption can be easily detected. This checksumming comes with additional overhead which negatively impacts performance. While we absolutely do not want to disable this checksumming in a real production environment it is possible to offload all this checksumming to the Lustre clients. This not only gets us true end-to-end checksumming which is desirable, but it removes this overhead from a single Lustre server and parallelizes it over your compute rich Lustre clients improving scalability. For this reason we felt that disabling checksumming was a fair way to simulate this performance improvement.

Next Steps: While we now have a working kernel port of the DMU there are still a few rough edges which need to be resolved.

- **Stack Usage**: By Linux kernel standards ZFS is stack heavy using roughly 10k of stack space. This is absolutely fine on Solaris which has 16k stacks, unfortunately on x86_64 Linux kernels the default stack depth is 8k. For now we have overcome this issue by building custom linux kernels with 16k stacks for development. To make this port stable for general Linux use the stack usage must be reduced so it can be used with precompiled Linux kernels.
- **Scatter/Gather**: To reduce the need for large contiguous memory allocations, and to increase the maximum DMU chunk size beyond 128k scatter/gather interfaces need to be added to the DMU. This should remove the need to do any manual tuning of 'vdev_max_pending' and 'zfs_arc_max'.
- **Lustre-ZFS Integration**: While the KPIOS benchmark provides a convenient interface for performance analysis the end goal is to interface the Lustre servers with the DMU. We must continue the work to interface our Linux kernel DMU port with a recent version of Lustre.
- **ZVOL**: Beyond the DMU which is strictly all that Lustre requires is a layer called the ZVOL. This layer sits on top of the DMU and provides a block interface to user space with all the benefits of a ZFS based file system. Completing this interface will provide a helpful user space access point to aid in debugging and performance analysis of the underlying DMU.
- **ZPL**: Additionally porting the ZFS posix layer, known as the ZPL, while not required would be a very helpful component for DMU based Lustre servers. Sun is currently planning to utilize the DMU in such a way that it will be possible to directly mount the ZFS based OSTs in an analogous manor to the current ldiskfs based OSTs. Porting the ZPL will allow easy data access on the Lustre servers. On a non-technical note, porting the ZPL will enable increased community support and backing for ZFS on Linux. A fully function port may also motivate Sun to dual license ZFS under the GPLv2 as they have expressed an interest in doing.

Appendix A:

Performance Summary						
Test	Write MiB/s	Read MiB/s	Write % of DD	Read % of DD	Write % Delta	Read % Delta
ZFS Baseline	570.76	611.1	53.57%	58.95%	0.00%	0.00%
ZFS Disable Checksums	602.57	610.54	56.55%	58.90%	5.57%	-0.09%
ZFS Disable Prefetch	557.68	822.08	52.34%	79.30%	-2.29%	34.52%
ZFS Simulated Zerocopy	673.66	615.18	63.22%	59.34%	18.03%	0.67%
ZFS pending=1024	602.05	743.91	56.50%	71.76%	5.48%	21.73%
ZFS arc=8g	600.92	682.65	56.40%	65.85%	5.28%	11.71%
ZFS (All Tunings)	845.66	991.06	79.36%	95.60%	48.16%	62.18%
DD Parallel Streaming	1065.54	1036.64	100.00%	100.00%	0.00%	0.00%