Proceedings of the Linux Symposium

Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.

MD RAID Acceleration

Support for Asynchronous DMA/XOR Engines

Dan J. Williams
Intel Corporation
dan.j.williams@intel.com

Abstract

The Linux MD driver performs RAID management operations entirely with software routines running on the CPU. This paper discusses the theoretical performance advantages of modifying MD's RAID-5 implementation to offload its operations to dedicated hardware resources.

1 Introduction

RAID (Redundant Array of Inexpensive Disks) host bus adapters are a common feature of server-class platforms. The core of these adapters is typically an embedded system-on-a-chip architecture comprised of a low power CPU paired with dedicated XOR calculation hardware and a memory controller. Firmware on this chip is tasked with taking a set of disks, combining them with a given RAID algorithm, and then exposing the RAID volume to the host operating system.

Today, system-on-a-chip I/O processor (IOP) designs are being reused in external storage applications, where the firmware is replaced with Linux. OEM's take advantage of this low cost hardware and Linux based storage software to produce entry-level NAS (Network Attached Storage) and SAN (Storage Area Network) appliances. The problem, however, is that the Linux MD driver does not take into account the dedicated RAID acceleration capabilities of an IOP. This paper discusses the implementation and benefits of modifying Linux's MD driver architecture to offload its compute intensive operations when in the presence of dedicated hardware (compute intensive operations include block xor, copy, compare, and zeroing). A corollary of this goal will be the capability for MD to spread operations over multiple CPUs in SMP configurations. It goes without saying that the performance of MD in systems without offload capabilities must not be significantly impacted.

The general approach of this work, as suggested by Neil Brown on the linux-raid mailing list [NB1], is to have MD queue block operations to a separate thread / resource. The following benefits arise from this approach:

1. Reduced CPU Utilization: Once an operation has been handed off to the backend hardware, the kernel is free to prepare the next RAID operation, causing a pipelining effect, or it can spend cycles in other areas of the system.

2. Speed improvements from hardware memory copies and XOR operations: The

architecture of the dedicated hardware units on an IOP is designed to process block operations at a higher rate than a software routine on the CPU

3. Reduced cache pollution: Some data involved in RAID calculations is never consumed by a client thread, or is temporally distant from consumption.

The overarching goal of these changes is to increase the appeal of Linux powered IOP platforms in the external storage appliances market by taking maximum advantage of IOP hardware in RAID array management.

2 Background

The IT industry's increasing intolerance for downtime paired with the continued growth of storage capacity requirements drives the market for RAID technologies. The RAID-5 and RAID-6 algorithms seek to increase fault tolerance while reducing the cost per unit of capacity compared to RAID-1/10 technologies. Server systems that implement RAID-5/6 typically do so with a RAID host bus adapter. Even though most server class operating systems can run the algorithms natively in the system kernel, users typically want to spend CPU cycles on transaction processing and leave storage subsystem maintenance operations for the adapter. In addition to offloading parity calculations adapters assume interrupt handling responsibilities for the disk controllers participating in the array. This, combined with large caches isolated from system RAM, allows the adapter to accelerate accesses to the disk array.

It naturally follows that when Linux is ported to run on the adapter processor, MD should be re-crafted to take advantage of the specialized hardware. An initial proof-of-concept experiment was attempted that made minimal changes to the current MD stack. However, it became apparent from this work that more fundamental changes to MD were needed in order to optimize the potential performance benefit of the offload engines. A description of the performance shortcomings of this first experiment and the potential benefits of the current work is best understood after a discussion of the current MD architecture.

3 MD RAID-5 Architecture Introduction

In essence, MD is a Linux block device driver that redirects incoming requests to a set of backing block devices that comprise the RAID array. There are no restrictions on the types of backing devices as long as they expose the Linux block I/O interface; in other words, beyond disks it is possible to use loop back devices, network block devices (NBD), or even other MD devices in a RAID configuration. Clients of an array submit a `bio`[1] directly to MD's request queue via its custom `make_request` method. This routine acts as front-end for the stripe cache and the stripe handling state machine. Compare this to standard disk block device drivers where `make_request` is the front end of one of Linux's I/O scheduler queues that coalesce and reorder the requests before submitting transactions to the disk. The stripe cache is a read-write-allocate and write-through cache of the blocks (pages) associated with backing disk data. It is worth noting that the stripe cache sits below the page cache in Linux's memory hierarchy increasing the chance that a client request can be serviced

[1] A `bio` is the unit of operation for the block I/O layer. A `bio` describes the source / destination sector of a transaction and points to a list of `bio_vecs` which can be thought of as a scatter-gather list [LKD].

by data in system RAM rather than suffering large latencies waiting for the disk to return the data. Before digging deeper into how the stripe cache gets populated and managed, a description of the key RAID-5 data structure, `struct stripe_head`, and the `handle_stripe` routine is required.

MD logically organizes the sectors of the backing disks into a series of `stripe_head`s. Each `stripe_head` contains an array of *N* `r5devs`, where *N* is the number of member disks and an `r5dev` is the data structure that, among other things, points to a `PAGE_SIZE` block of data that it is the cached version of data on the backing device. The following is a full description of the `r5dev` structure members:

- `struct bio req`: `bio` reserved for sending transactions to the backing disk
- `struct bio_vec vec`: `bio_vec` reserved to describe the location of data for transactions to the backing disk
- `struct page *page`: The stripe cache version of disk data
- `struct bio *toread, *towrite`: Lists of incoming requests queued via `make_request`
- `struct bio *written`: Queue for write transactions that have hit the stripe cache but now need to be propagated to the backing disk
- `sector_t sector`: Logical sector of the array that the `page` pointer references
- `unsigned long flags`: Bit field that contains the state of the cached buffer, and flags for requesting service like `R5_Wantread` and `R5_Wantwrite`

As mentioned the `r5dev` array is a member of the `stripe_head` data structure. The pertinent members of this structure required for the stripe cache are:

- `sector_t sector`: Logical array sector associated with the `stripe_head`
- `int pd_idx`: Which device in the `r5dev` array holds the parity information for this stripe. The parity information rotates according to a separate algorithm; by default this is left-symmetric. In the RAID-4 case the parity disk is constant across the entire array.
- `unsigned long state`: State of the stripe that tracks whether the stripe needs servicing, whether it is waiting for data before a transaction can continue, or whether it is part of a secondary operation like synchronizing or expanding.
- `atomic_t count`: This reference counter logs the number of individual threads that are currently operating on the stripe. Once each thread has had a chance to log its requests, the `stripe_head` is passed to the `raid5d` kernel thread for further handling.
- `spinlock_t lock`: Serializes access to the contents of the `stripe_head`.

A `stripe_head` begins life when a transaction, received via `make_request`, targets the physical disk sectors of its backing disks. It ends life when its reference count drops to zero (no client threads are performing operations) making it a candidate for stripe cache victimization.

4 MD RAID Acceleration

The state transitions and operations that a `stripe_head` undergoes on its path from activation to victimization are the focus of the present acceleration work. The current flow for a write operation follows; the description assumes that we are overwriting complete blocks in the stripe resulting in a 'reconstruct-write' operation.

1. A thread submits a `bio` with data to be written to `make_request`. `make_request` calls `add_stripe_bio` which checks to make sure that a request is not already pending before queuing the new request. In the case that a request is already pending the thread is put to sleep, and subsequently woken once the preceding request has been submitted.

2. With a new `bio` posted to the stripe `make_request`, still in the context of the requesting thread, calls `handle_stripe` to start the raid5 state machine. Assuming this is the first operation to the stripe, `handle_stripe` will determine that it needs to fill the cache with all the other blocks in the stripe in order to perform the xor parity calculation. Once `handle_stripe` submits the read requests to the backing disks it hands over further stripe operations to `raid5d`.[2]

3. Once all of the reads to disks have completed `raid5d` is woken up to advance the state of the stripe and perform the 'reconstruct-write' operation. A reconstruct-write involves copying data from the `bio` (submitted in step 1) into the relevant blocks, zeroing the parity block, and finally regenerating parity across all the blocks in the stripe. It is important to note that all of the work in this step is performed under a per stripe lock.[3] The written blocks and the updated parity block are now submitted to the backing disks.

4. `raid5d` is woken up again when the writes complete. It signals completion of the original write request (submitted in step 1) by calling the `bi_end_io` method of the `bio`. Assuming no new requests have been submitted to this stripe it is transitioned to an inactive state and `raid5d` returns to sleep.

The acceleration work targets the operations of moving data out of the stripe cache,[4] maintaining parity, and recovering degraded blocks from the other disks. These operations entail memory block copying, block xor, block xor with a 'zero-result' check (to verify parity), and block fills (to prepare parity blocks that will be overwritten). On an I/O processor, like the IOP333, all of these operations can be carried out in hardware. Initial proof-of-concept work along these lines helped to illuminate the form of the final design. The experiment involved making in-line replacements of the software xor routine with a routine that set up and ran the operation with the hardware unit; however, it did not show significant improvement gains. As Neil Brown said on the linux-raid mailing list, "I'm not surprised that simply replacing `xor_block` with calls into the hardware engine didn't help much. `xor_block` is currently called under a spinlock, so the main processor will probably be completely idle while the AA is doing the XOR calculation, so there isn't much room for improvement." [NB1]

[2] `raid5d` is a thread that collects and executes array maintenance operations, there is one thread for each array in the system.

[3] The stripe lock is a per stripe `spin_lock` that prevents other threads from concurrently modifying the stripe's state in the `handle_stripe` routine.

[4] Moving data into the stripe cache is handled by the block drivers of the backing devices.

Performing the operations under a spin lock in `raid5d` prevents pre-emption, precludes multiple operations being queued to the same stripe, and from a system throughput perspective stalls the stripes waiting in `raid5d`'s queue. The first phase of the current acceleration work is to move the block operations outside the `stripe_head` lock, and outside of `raid5d`. The current patches achieve this by submitting the operations to a kernel work queue. This has the side benefit of allowing SMP systems, without xor and copy engines, to spread RAID work over multiple CPUs. One of the effects of using the `raid5d` thread to shepherd the `stripe_head` through its states of operation is that work that is parallel in nature is processed in a single threaded fashion. SMP systems with the work queue changes will be able to accept work from multiple threads and then disperse that work across all the CPUs in the system. In the uni-processor IOP case, the work queue will interface with an offload API to submit batches of operations to hardware engines. The I/OAT DMA engine API [IOAT], presented at last year's OLS 2005, is the basis for this acceleration interface.

In the presence of hardware offload engines, the work queue will be responsible for de-queuing requests as fast as possible while maintaining the order of transactions. The order of transactions is important because RAID operations, like the write case outlined earlier, require several steps that must complete in order. For example, consider a case where the work queue submits the drain operation (copy from bio to stripe cache) for a write, and subsequently advances to the stage that performs the xor across the new data in the stripe. If the xor starts before the copy completes, the parity becomes corrupted. It is possible to envision systems with several flavors of hardware offload configuration; some may have an all in one unit that can handle all the block operations from a single queue, while some may have several discrete units, one per operation type. A given work queue thread implementation is required to detect and manage cases where an operation must be stalled until a hardware unit completes. Again, this is not necessary if the hardware can maintain ordering.

The `raid5d` thread in this new model is now only tasked with finding work, submitting work, and advancing the state machine. This new arrangement aims submitting work, and advancing the state machine. Block operations are queued to a separate, to reclaim CPU cycles spent performing block operations and allocate them to other system tasks. With hardware acceleration this model also enables some configurations to entirely eliminate the CPU data cache footprint of the stripe cache. Consider that high performance disk adapter drivers arrange for data to be directly pushed and pulled from memory by the device (bus mastering). With this capability stripe data enters the stripe cache via a device to host write operation, is maintained in the cache via DMA engine xor operations, and leaves the cache by either a device to host read operation or a DMA copy to `bio` operation, never dirtying cache lines in the CPU. The need to repetitively synchronize the stripe cache with main memory was a contributing factor to the sub-optimal performance of the proof-of-concept experiment.

5 Conclusion

MD driver block operations can be accelerated by taking full advantage of the hardware features of an IOP. To date, patches have been submitted to move the block operations to a work queue and the next steps are to integrate the dmaengine API into the work queue. Once the RAID-5 changes have settled, the same approach will be applied to the RAID-6 layer where parity manipulation can be significantly

more compute intensive. This work, while primarily beneficial to embedded systems with hardware-offload engines, also has the potential to enhance MD performance on SMP systems.

References

[NB1] Neil Brown, *Re: Accelerating Linux software raid*, Electronic Communication, September 2005, `http://marc.theaimsgroup.com/?l=linux-raid&m=112648052213893&w=2`

[NB2] Neil Brown, *Re: [RFC][PATCH 000 of 3] MD Acceleration and the ADMA interface: Introduction*, Electronic Communication, February 2006 `http://marc.theaimsgroup.com/?l=linux-raid&m=113988782117987&w=2`

[LKD] Robert Love, *Linux Kernel Development, Second Edition*, 2005

[LDD] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, *Linux Device Drivers, 3rd Edition*, 2005

[IOAT] Andrew Grover and Chistopher Leech, *Accelerating Network Receive Processing: Intel I/O Acceleration Technology*, Linux Symposium, 2005, `http://www.linuxsymposium.org/2005/linuxsymposium_procv1.pdf`