Optimizing a Highly Fault Tolerant Software RAID for Many Core Systems

Henning Klein
Fujitsu Technology Solutions GmbH
Henning.Klein@ts.fujitsu.com

Jörg Keller
Fernuniversität in Hagen
Joerg.Keller@FernUni-Hagen.de

ABSTRACT

We present a parallel software driver for a RAID architecture to detect and correct corrupted disk blocks in addition to tolerate disk failures. The necessary computations demand parallel execution to avoid the processor being the bottleneck for a RAID with high bandwidth. The driver employs the processing power of multicore and manycore systems. We report on the performance of a prototype implementation on a quadcore processor that indicates linear speedup and promises good scalability on larger machines. We use reordering of I/O orders to ensure balance between CPU and disk load.

KEYWORDS: Fault-Tolerant Software Design; Multi-Core RAID Architecture

1. INTRODUCTION

RAID architectures [1] have been introduced to tolerate disk failures. It turns out however, that also functional disks over time evolve corrupted data blocks by so-called silent errors [2]. This type of error gains in importance with the advent of solid-state disks due to wear out of memory cells. Although SSDs keep an additional space to replace defective cells it's only a matter of time and type of usage until this space gets insufficient. Recent developments showed that the reliability of SSDs is mostly sacrificed for capacity and cost per bit [9]. Especially the process of decreasing cell sizes, while increasing the number of storable states rose concerns about the reliability. Other than that SSD cells are not only getting worn out due to write processes, they tend to lose their state over time as well. While there exist checksum schemes such as ZFS [3] to handle silent errors, the overhead is notable, and hence integration of schemes for disk failure and data block corruption seems advantageous.

The Software RAID considered in this paper was introduced in [4] and is organized as follows: Each disk is divided into equally sized blocks of n Sectors, starting at a certain offset. All blocks across the disks in the array at the same position are combined to a stripe. Every stripe contains two blocks storing parity values, whereas the computation of each parity value ranges over two stripes. We denote the four parity values within two stripes by P, Q, R and S. The values computed are based on Reed Solomon codes using multiplications with Generators g based on finite fields, see Fig. 2. By combining the four parity values of two stripes as depicted in Fig. 1, detection and correction of corrupted disk blocks is possible in addition to RAID-6 behavior in case of disk loss. However, to detect corrupted blocks, parity values must be recomputed with each read and compared to stored parity values. The architecture has been optimized by using optimized lookup tables to speed up computations [5], but the resulting computational effort is still high. Moreover, with a larger number of disks providing data at high bandwidth, the computation itself could turn out to be the bottleneck. Previous investigations were primarily focusing on how to speed up encoding speeds during writes. This paper shows, how such an architecture could be implemented effectively in a RAID driver that uses parallel threads to exploit computational capabilities available in multicore processors, to balance disk bandwidth and computation. The driver has to cope with different priorities of read and write orders which renders low-overhead parallelization non-trivial. Some RAID systems like RAID-5 or RAID-6 have already been implemented in the major operating systems Linux [6] and Windows. The Linux implementation focuses on exploiting special instruction sets like SSE or MMX [7] rather than taking advantage of multiple processors. The most complex RAID configuration that Windows supports is RAID-5, which is implementable delivering a sufficient speed using a single core only, however the implementation remains undocumented. There have been investigations on speeding up Reed/Solomon-encoding with a FPGA-based coprocessor [8] which requires additional hardware.

	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
Stripe 0	0	1	2	P0	R0
Stripe 1	3	4	5	Q0	S0
Stripe 2	6	7	P1	R1	8
Stripe 3	9	10	Q1	S1	11

Figure 1. Organization Of Data Blocks In The RAID

$$P = D_0 + D_1 + \cdots + D_n$$
$$Q = g^0 * D_0 + g^1 * D_1 + \cdots + g^n * D_n$$
$$R = g^0 * D_0 + g^2 * D_1 + \cdots + g^{2n} * D_n$$
$$S = g^0 * D_0 + g^3 * D_1 + \cdots + g^{3n} * D_n$$

Figure 2. Parity Calculations

The remainder of the paper is organized as follows. Section 2 describes the architecture of a parallel RAID driver and describes an optimized reordering to enable parallel disk and parity processing. In Section 3 we show the prioritization concept of the threads and present figures showing the advantage of connected disk orders. In Section 4 we give a conclusion and an outlook on future work.

2. RAID ARCHITECTURE WITH OPTIMIZED REQUEST REORDERING

The RAID architecture performs integrity checks when reading and generates parity when writing blocks. Therefore each disk I/O operation involves parity computation. Both of the steps to process read and write operations are parallelized to gain speedup and they can further be improved by implementing them parallel to each other, if possible. If parity checks fail, a recovery process has to be started to recover corrupted data. All data elements in two stripes have to be known to generate or check data. Thus, if read or write operations start or end in the middle of two stripes, the remaining data has to be read from the disk array first. In case of write orders, this data has to be checked first, in order to avoid the generation of parity with possibly corrupted data. This can only be achieved, if the whole dataset of two stripes will be read, even the part that will be overwritten later. From a performance perspective, reading full double stripes won't generate too much overhead, as data blocks will be read in parallel from all disks and block sizes are chosen in a range so that reading any smaller amount wouldn't be significantly faster because of the comparatively higher disk access overhead. The whole process is depicted in Fig. 3.

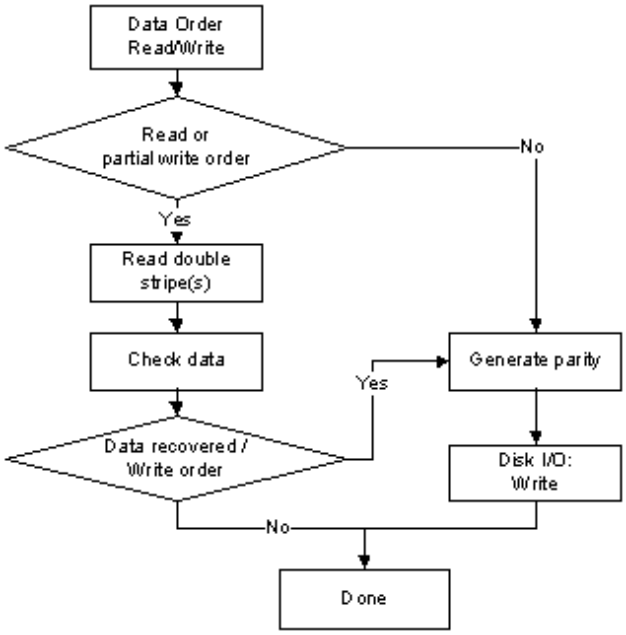

Figure 3. Parity And Disk Access Diagram

2.1. Parallelizing Parity Computation

Parity is computed in parallel threads in order to avoid it from being a bottleneck and to gain the fastest possible response time if computations have to be done serially to disk I/O operations. We generate parallel processable work loads by splitting up data blocks, in a way that each execution path accesses each part of all blocks with the same offset within two stripes and generates full sets of parity values, see Fig. 4. With this method the accessed data is being separated between each thread without the need of further synchronization operations to perform simultaneous write accesses to the same data blocks. Secondly it's easy to generate equally sized workloads for almost any number of parallel execution paths. Nevertheless we've considered splitting up the data by assigning whole blocks, see Fig. 5. To compute parity we use lookup tables holding precomputed multiplication results. When computing parity, a different part of the lookup tables is being used for each data block. By assigning whole blocks, each thread and processor would only have to access parts of the lookup tables which lowers cache misses. However, by optimizing the table access order as described in [5] the first method is producing the best results while providing better scalability.

	Disk 0		Disk 1		Disk 2		Disk 3		Disk 4	
Stripe 0	T1	T2	T1	T2	T1	T2	P: T1	P: T2	R: T1	R: T2
Stripe 1	T1	T2	T1	T2	T1	T2	Q: T1	Q: T2	S: T1	S: T2

T1: Thread 1, T2: Thread 2

Figure 4. Assigning Parts Of All Blocks To Threads

	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
Stripe 0	T1	T1	T2	P: T1/T2	R: T1/T2
Stripe 1	T1	T2	T2	Q: T1/T2	S: T1/T2

T1: Thread 1, T2: Thread 2

Figure 5. Assigning Whole Blocks To Different Threads

2.2. Optimizing Write Orders

Write orders are easier to optimize than read orders, as they can be buffered and reported back to the caller as completed although they are not even started. While the buffer is being filled with write operations, the storage driver can optimize the writing process by reordering the commands and parallelizing calculations and I/O operations. If the buffer reaches its limit, one of the resources, usually the disk, will be the limiting factor, blocking upcoming orders. The number of disk I/O operations can be reduced by connecting consecutive write operations, see Fig. 6, which shows the merging of data blocks of Fig. 1. Writing consecutive data blocks will result in a higher I/O speed. Secondly, the number of necessary synchronization operations can be reduced as well. Each thread would normally compute one block at a time and then synchronize to receive a new order from the queue.

Thread a					Thread b				
0a 6a	1a 7a	2a 8a	P0a P1a	R0a R1a	0b 6b	1b 7b	2b 8b	P0b P1b	R0b R1b
3a 9a	4a 10a	5a 11a	Q0a Q1a	S0a S1a	3b 9b	4b 10b	5b 11b	Q0b Q1b	S0b S1b

Figure 6. Combining Blocks To Reduce Synchronization

If write orders are buffered, the worker thread can be supplied with the number of unprocessed orders instead of synchronizing for every single order. Using connected blocks will help ordering the overhead access to the lookup tables for finite field multiplications, which reduces the number of CPU cache misses. That way the overhead can be reduced and a bunch of smaller I/O operations will be computed more efficiently on multiple cores.

2.3. Optimizing Read Orders

Read orders cannot be buffered as the calling instance could probably use the data immediately after receiving a completion receipt. Without read ahead techniques, which is the case if random read accesses occur, the design goal has to be reacting as fast as possible. The response time will be the sum of parallel disk I/O and parity checking. To enable buffering techniques a read ahead should be performed immediately after the requested block has been transferred, to utilize the disks while parity is being checked. The amount of data that will be read depends on many factors: the size of the preceding read command, the time it needs to be processed, the characteristics of the storage medium and the probability of the used data to be requested next.

2.4. Optimizing Mixed Read And Write Orders

In an optimal case all of the upcoming read and write operations are known and the RAID driver is able to parallelize disk and CPU load. Depending on the speed and the number of available resources either parity processing or disk I/O operations would limit the bandwidth.

For write operations this can simply be achieved by buffering orders. Read operations cannot be buffered and therefore they block upcoming orders until they are processed. If the driver receives alternating read and write requests without rescheduling their order it would have to process disk I/O and parity computations serially which dramatically decreases performance. To avoid read operations from emptying the buffer, they have to be processed with a higher priority. In this case, they will interrupt the processing of buffered write operations and can seamlessly utilize CPU and disk bandwidth. As long as there is space in the buffer, upcoming read operations will stall the write operations.

Two important optimizations, as shown above, are working against this scenario of mixed read and write commands. If multiple disk write orders are merged to reduce synchronization overhead, a single read operation cannot be placed in between. If read operations interrupt consecutive write orders, the additional positioning times of rotating disks could decrease the disks' I/O performance. Newer solid state disks however would be less impacted, although they are profiting of consecutive I/Os as well. Therefore a lot of tuning has to be done that has to fit the type and number of used storage media and the performance and number of processors.

Depending on the application the size and rate of the read and write operations will vary. A database will create a lot more small and random accesses than a storage server. Random read accesses will result in unsuccessful read aheads. If the database doesn't request parallel read operations, the parity checks will have to be done serially to the disk I/O. Therefore the optimizations are restricted to more general types of usage involving mixed read and write requests.

3. IMPLEMENTING THE RAID DRIVER

3.1. Thread Architecture

To accomplish fast reaction for small read operations and high throughput for write operations the computations and disk I/O commands are being split in different threads with separate order queues. The order enqueuing function checks whether the next order can be combined with previous ones to reduce overhead. The optimal maximum combined order size depends on the number and types of disks used.

One thread is implemented for each processor or core, to parallelize parity computations. To ensure that parity computations for higher priority read operations are processed as soon as possible a second set of threads is started with a higher priority. The scheduler of the operating system will switch to the higher priority threads as soon as orders are enqueued. The highest priority thread in the driver will be used for disk I/O threads, to prevent them from being blocked by parity generation threads. The enqueuing of orders will be prioritized above parity generation of write operations to ensure they won't be blocked when connecting orders, see Tab. 1.

Table 1. Assigning Priority Boosts To Threads

Priority Boost	Thread Type
3	**Disk I/O: RW**
2	**Parity Check (Read)**
1	**Enqueue I/O**
0	**Parity Generation (Write)**

To simplify the prioritization of read over write orders, which usually are placed one at a time, a single variable is used to signal a read order. This variable will be checked by the disk thread before an order is used from the regular queue of write orders. If no orders are available in the queue after completing the read operation, a read ahead can be started to utilize the storage media while parity is being checked. This enables parallelization of computations and disk I/O like it's being done with buffered write operations.

3.2. Parameter Consideration

Depending on the number of disks and the block size that is used, the merging of orders that reduces overhead is more or less significant for the overall computation speed. If computations can be done in parallel to and in less time than disk I/O orders, only more processor time would be consumed. Otherwise, e.g. in case of random read orders, it will directly affect the performance of the driver. Fig. 7 and Fig. 8 show how the number of disks and the block size affect the computation speed on single and multiple processors.

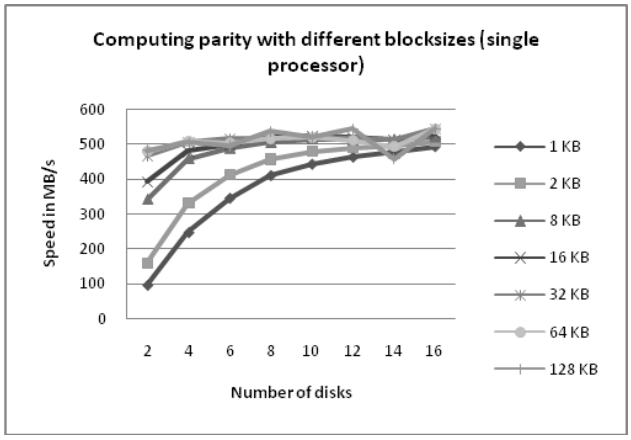

Figure 7. Computing Parity Of Different Block Sizes On A Single Processor

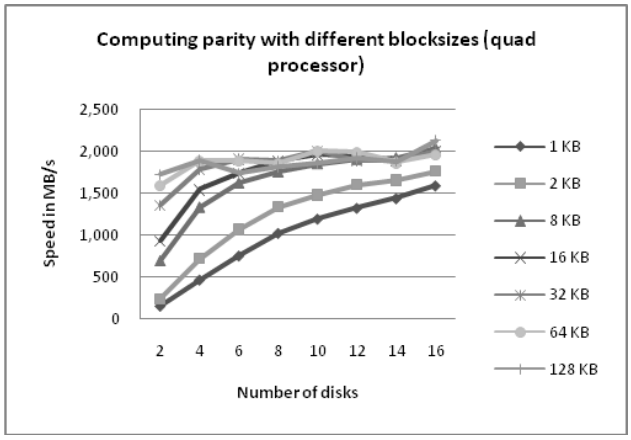

Figure 8. Computing Parity Of Different Block Sizes On Four Processors

Having a look at the disk write speeds of rotating and solid state disks one can see that the transfer speed of consecutive blocks can be increased if connected and

submitted as one order. The performance on rotating disks is significantly higher with connected blocks, see Fig. 9. The response time and transfer speeds of consecutive read orders indicate that reading blocks below a certain size won't be effective. With solid state drives this amount can be decreased, as they reach the transfer limit sooner than rotating disks. This will reduce the performance decrease of wrongly predicted read aheads, see Fig. 10 and Fig. 11.

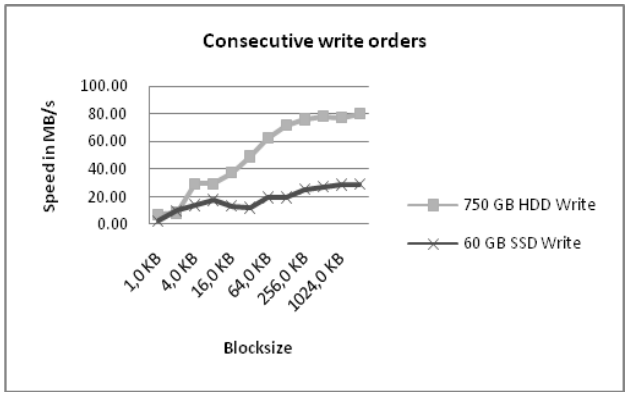

Figure 9. Speed Of Consecutive Write Orders

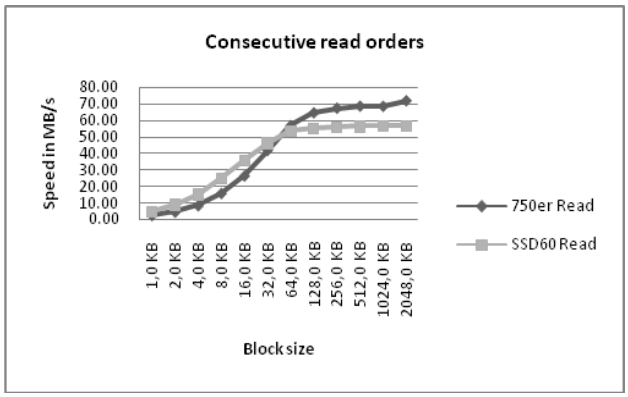

Figure 10. Speed Of Consecutive Read Orders

4. CONLUSIONS

We have presented the design of a parallel RAID driver that exploits multicore capabilities to achieve high-speed fault-tolerant disk access, not only in the case of a disk failure, but also in the case of data block corruption. We reported on the performance of a prototype implementation that demonstrated good speedups and hence scalable throughput. The presented figures about single and quadcore processor performance show almost linear speedup for most block sizes, which indicates that the architecture will scale well on a higher number of processor cores. Comparing today's disk speeds with the results of the computations on a consumer grade PC we are confident that the proposed architecture will suffice for RAID configurations with current and future hardware.

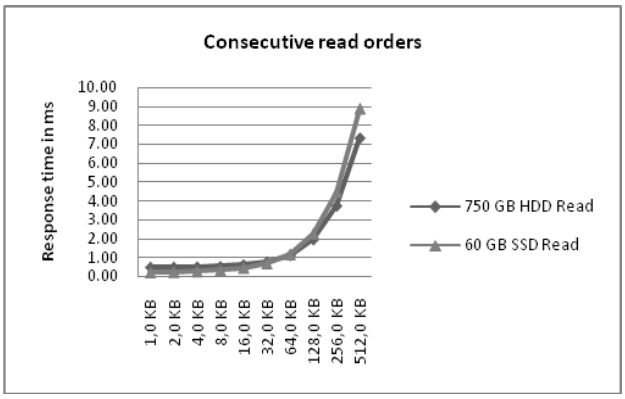

Figure 11. Response Time Of Consecutive Read Orders

Future investigations will focus on tuning the schemes evolved towards applications such as long-term archival storage, where the number of silent errors, the time to disk failure, and the number of test reads to correct the former have a complex relation that must be optimized to achieve reliable long-term storage with a competitive effort. A different approach to speed up the RAID for applications using many random read operations, such as databases, is reducing the parity checking to on demand self correction, i.e. during rebuild or verify operations. Another aspect could be adjusting the number of used CPU cores depending on the current computational workload generated by other applications.

REFERENCES

[1] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in SIGMOD '88: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, 1988, pp. 109–116.

[2] J. Bonwick and B. Moore, "ZFS — the last word in file systems," http://opensolaris.org/os/community/zfs/docs/zfs last.pdf.

[3] Sun Microsystems, "Sun on-disk specification, "http://opensolaris.org/os/community/zfs/ docs/ondiskformat-0822.pdf.

[4] H. Klein and J. Keller, "RAID architecture with correction of corrupted data in faulty disk blocks" in Proc. 6th ARCS Workshop on Dependability and Fault-Tolerance, Delft, NL, Mar. 2009.

[5] H. Klein and J. Keller, “Storage Architecture with Integrity, Redundancy and Encryption” in 14th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS '09), Rome, May 2009.

[6] Linux Kernel sources – md driver, Version 2.6.29.1, April 2009, http://kernel.org

[7] H Anvin, “The Mathematics of RAID-6”, 2009, http://www.kernel.org/pub/linux/kernel/people/hpa/raid6.pdf

[8] V. Hampel, P. Sobe, and E. Maehle, “Experiences with a FPGA-based Reed/Solomon-encoding coprocessor,” Microprocess. Microsyst., vol. 32, no. 5-6, pp. 313–320, 2008.

[9] Silicon Systems, Inc., “NAND Evolution and its effects on Solid-State Drive (SSD) usable life”, http://www.siliconsystems.com/technology/pdfs/WP-001-00R.pdf