Providing Single I/O Space and Multiple Fault Tolerance in a Distributed RAID

Alessandro Di Marco, Giovanni Chiola, Giuseppe Ciaccio
DISI, Università di Genova
via Dodecaneso, 35
16146 Genova, Italy
{dmr,chiola,ciaccio}@disi.unige.it

Technical report DISI-TR-03-06

27 May 2003

Abstract

Commodity EIDE disks provide low cost storage but are severely limited in bandwidth and cannot be made fault-tolerant. On the other hand, conventional RAID devices provide reliability and performance but worse price/performance figures.

A cluster of PCs can be seen as a collection of networked low cost disks; such a collection can be operated by proper software so as to provide the abstraction of a single, larger block device. By adding suitable data redundancy, such a disk collection as a whole could act as single, highly fault tolerant, virtual RAID device, providing capacity and reliability along with the convenient price/performance typical of commodity clusters.

Such a distributed RAID could even be accessible by each of the stations in the cluster, rather than throughout one or few end-points, with a potentially higher aggregate I/O bandwidh and better suitability to parallel I/O. The overall picture is a prototype of a so-called "I/O-centric cluster", namely, a cluster of PCs offering storage services under a Single I/O Space (SIOS), rather than offering traditional computing power.

This paper reports about the design and performance of a prototype of such a virtual RAID system, called DRAID, on a Gigabit Ethernet cluster of PCs.

Keywords: RAID, network-attached storage, cluster computing, parallel I/O, single system image.

1 Introduction and motivation

The information age we live in poses increasingly hard requirements on storage systems. The need for fast and reliable storage is imperative in many mission-critical frameworks, from corporate to medium-size enterprise to scientific experiments, in which data rate requirements in the range of GByte/s are often coupled with very strong availability assumptions. But also the cost of the processing and storage infrastructure is a critical parameter.

At the low end of the spectrum of storage technologies we currently find conventional EIDE/SCSI disk drives. Such low-cost commodity devices have shown tremendous improvements in terms of cost/capacity but a much less satisfactory increase of performance compared to other system components. For example, most CPUs can write to DRAM at 2 GByte/s bandwidth and fiber optics links can pipe data at 10 GByte/s, whereas the exploitable throughput of conventional EIDE disks still lag far behind a modest 100 MByte/s, not to mention their known lack of robustness and fault tolerance.

At the high end of the spectrum, however, we find storage devices providing higher capacity together with much greater performance, thanks to techniques known as data declustering and disk striping [14, 13] which allow some amount of parallelism across many disks. However, incorporating a large number of disks into a disk array makes the storage system more prone to failure than a single disk. Higher reliability is then achieved by using redundant encodings of data so as to survive one or more disk failures, yielding what is called a RAID [10]. Most RAIDs are capable of tolerating the failure of one of their disks; greater fault tolerance can be obtained by exploiting more sophisticated error correcting algorithms, usually based on Hamming or Reed-Solomon codes [11].

Hardware RAIDs provide a better answer to the need for fast and dependable storage, compared to single hard disks. Their main disadvantage is a far worse cost/capacity ratio (with the exception of the entry-level RAID-1 systems often found on small workstations). Other disadvantages come from the centralization of resources into a single package, which increases the risk of unrecoverable failure in case of localized accident (a fire, for instance), and a non negligible correlation among single disk failures within the RAID, which is inherent in the production process of RAIDs themselves and leads to greater probability of multiple, simultaneous and thus unrecoverable disk failures, compared to an equivalent system made up of stochastically uncorrelated disks.

Large RAIDs have become ubiquitous inside those devices like file servers and network-attached storage systems, which are to provide services to communities of users. The most typical setting is one in which a centralized (and expensive) RAID-based common server is shared by a number of workstations or Personal Computer (PC) with small local hard disks, throughout a high bandwidth Local Area Network (LAN). The overall architecture is thus a cluster of PCs with a shared, high-performance storage server as a central, and expensive, core.

However, as each PC has a local disk which is largely unused, one might think that aggregating and then sharing the storage capacities locally available at the individual stations could provide an alternative answer to the need for collective, high-capacity and reliable storage. It might be worth thinking at a capable, dependable, and cost-effective storage system made out of a collection of inexpensive EIDE disks, in which proper software implements RAID techniques by striping and replicating data over multiple disks throughout the existing networking infrastructure. The system could provide the abstraction for a single, huge "virtual disk", hiding the address spaces of each individual EIDE disk in the cluster by only using a small fraction of the cluster processing power to perform address translations and data striping/replication. This software approach offers the great advantage of allowing arbitrarily sophisticated techniques for data striping and redundancy, suitable to run-time adjustment; that is, a far greater flexibility in trading speed and space for reliability. A distributed RAID is easily expanded, up to saturation of the interconnection bandwidth, by just adding more PCs to the cluster. The peer-to-peer architecture of a distributed RAID leads to greater availability, thanks to the absence of single points of failure, lower correlation among disks, and absence of centralization within any single package. Last but not least, this approach tends to preserve the low cost/capacity ratio typical of commodity EIDE disks.

The idea sketched above is certainly not a new one. A system like the above is usually referred to as a *distributed RAID* [15], or "Do-It-Yourself RAID" [2]. However, early works on the subject only focused on the opportunity for better cost/capacity. Only a decade later it emerged that these systems could offer another, important advantage over classical RAIDs. Indeed, the abstraction of a single block device out of a collection of physically distinct disks is provided by cooperation among the involved stations. If such cooperation takes place on a peer-to-peer basis, each involved station can be given equal access to the common abstract device regardless of the location of physical storage resources, a feature now called *single I/O space* (SIOS) [6] which is a special case of a more general feature called *single system image* [12]. Now, providing SIOS at cluster level has an important implication, namely, it allows simultaneous access to the distributed RAID from multiple end-points, as all stations can actually serve as end-points to the virtual device. This yields more potential for higher aggregate throughput and better suitability to parallel I/O, compared to a classical file server or network-attached storage system.

2 DRAID: A Distributed RAID based on Reed-Solomon

2.1 Reed-Solomon basics

A Reed-Solomon error correcting code takes a sequence S of elements from an alphabet and produces another sequence S_r on the same alphabet. S_r is longer that S because some data redundancy is introduced in the representation. The advantage of a Reed-Solomon encoding comes from the possibility of recovering from *cancellations*, that is, missing elements in S_r. Indeed, with Reed-Solomon one is allowed to recompute the missing elements in S_r from the remaining ones, up to some extent, provided that the position in S_r of the missing elements be known. The maximum number of recoverable cancellations is a parameter of the encoding: more redundancy means greater size of S_r compared to the original sequence, but also more tolerance to cancellation.

We refer to [11] for the mathematical details of Reed-Solomon codes, here limiting ourselves to a simple and informal example. Let us consider a sequence $S = D_1D_2...D_N$ of size N. According to a simple Reed-Solomon encoding, we can compute a new sequence $S_r = D_1D_2...D_NP_1P_2...P_K$ of size $N + K$, which extends S by the addition of more elements. Each of the P_i new elements is computed as a linear combination of the $D_1, ..., D_N$ elements, in a suitable algebra of the alphabet symbols. If the algebra of the alphabet symbols is a field, and the linear combinations selected for computing each of the P_i are independent from one another, then we can choose any K elements from S_r and write each of them as independent linear combination of the remaining ones. So, we are allowed to compute any K missing elements of S_r from the remaining $N - K$ ones, regardless of the missing/remaining elements being Ds or Ps.

2.2 Reed-Solomon and distributed RAIDs

Cancellations can be a suitable failure model for a distributed disk array. Let us consider a traditional workstation with its own hard disk. The workstation runs an operating system which cooperates with the disk controller to provide a uniform failure model for the underlying hard disk: a so-called "I/O error" means that a read/write operation on the local disk has failed and, in the case of a read, no data are available. This is meant to avoid that a broken disk could silently provide wrong data (no data is better than corrupted ones). Now, let us consider a distributed disk array formed by an ordered collection of stations. A write operation on the array takes a block of data and stripes it across the stations in parallel. Then, a read operation of the same block obtains data back as an ordered sequence of the individual stripes coming from the accessed stations/disks. A not responding station as well as a not responding or broken disk as well a generic connectivity trouble can result in one out of a lot of different error messages ("I/O error", "connection timed out", etc.), but in the end all of them end up with a missing stripe in the block, which can be modeled as a missing element in a sequence, that is, a cancellation. [1]

Since errors appear as cancellations, Reed-Solomon can be taken into consideration to turn a distributed disk array into a RAID. Multiple failures are very likely in such a distributed environment, so we need codes capable of correcting multiple errors. Reed-Solomon codes are a good choice for distributed RAIDs because of their favourable aspect ratio between the desired degree of fault tolerance and the extra space needs posed by redundancy.

2.3 Architecture of DRAID

DRAID is a virtual block device that distributes blocks across workstations in a cluster, by first computing a Reed-Solomon redundant encoding of each block, then striping the resulting data among disks (or any other block device local to the the stations).

Given two natural numbers N and K, each logical block B is partitioned into a sequence of N segments $B_1B_2...B_N$, which is then extended with additional K redundant segments $P_1P_2...P_K$ computed by a Reed-Solomon encoder. All the Bs and the Ps have same size. The resulting *data stripe*, $B_1B_2...B_NP_1P_2...P_K$, counting as many as $N+K$ segments, is striped across $N+K$ stations/disks on a write operation. A subsequent read operation of the same logical block only needs to get *any* N out of $N+K$ segments from the data strip, in order to be able to successfully deliver the entire logical block $B_1B_2...B_N$.

From the above, it follows that K amounts to the degree of fault tolerance. K could be in principle chosen arbitrarily large. However, as $N+K$ is actually related to the degree of parallelism in the read/write operations, the optimal values for K and N depend on various other parameters (logical block size, communication and disk performance characteristics). Another consequence is that the minimal configuration of DRAID must count $N+K$ workstations, grouped together into what we call a single *stripe* of the RAID.

A realistic distributed RAID might count quite a lot of disks, in the order of hundredths. Clearly we cannot organize all disks into a single stripe, as we cannot expect the degree of parallelism $N+K$ to be that large (with fine grained parallelism, read/write performance would be dominated by the non-negligible latency incurred by operations on physical disks). Besides, adding more disks to an already working stripe would require too high an overhead, as changing N and K would require recomputing all the data strips. To allow expanding the DRAID at run time, we organize the set of stations/disks as a two-dimensional matrix, in which each row is a stripe of fixed size, counting exactly $N+K$ disks. This way, a DRAID can be expanded at will with the only constraint of adding disks in multiples of $N+K$.

The current prototype of DRAID is operated by a peer-to-peer software in the form of a kernel module

[1]Data corruptions coming from the network are masked off by numerous error checks along the communication protocol stack.

for Linux 2.4. It does not require modifications to the kernel. Communications towards remote disks are accomplished through UDP sockets, accessed "from beneath", that is, from kernel directly. The DRAID module operates the local disk of each workstation at the level of Linux logical block device, that is, the Linux buffer caches associated to the local disks are successfully exploited rather than barely bypassed.

From the software point of view, the peer-to-peer nature of DRAID, with the absence of any centralized data structure or control task, contributes to the robustness of the system as a whole. DRAID provides a SIOS abstraction, as all workstations are provided with the same logical view of the entire virtual block device regardless of their physical placement in the cluster. Each station can act as independent end-point to the logical block device; this makes DRAID potentially interesting for parallel I/O in a cluster environment.

2.4 Optimizations

Communication performance is notoriously poor when short messages are sent through sockets on a traditional LAN, due to the large impact of communication latency over the total delay. Messages carrying data blocks from/to disks pay a even larger penalty, due to the huge latencies of disk operations. As a consequence, acceptable levels of performance could only be attained by clustering together those block requests which happen to be contiguous in the address space of the virtual block device. We call a *block cluster* any sequence of contiguous blocks in a block device.

In Linux, each request to a given block device is actually issued to a *buffer cache* associated with the block device, and this indeed occurs with the DRAID device too. The Linux buffer cache manager provides a convenient interface between the block device abstraction offered to applications and the kernel-level buffer cache, as well as between the cache and the physical device.

By cooperating with the Linux buffer cache manager, DRAID is able to take advantage from the possible clustering of distinct read/write requests to contiguous virtual blocks. Of course, the positive effects of these optimizations are maximized when the user applications read/write quite long sequences of contiguous virtual blocks.

2.4.1 Write clustering

On write, the current prototype of DRAID supports *write clustering*. The Linux buffer cache manager reorders the virtual blocks written in the buffer cache, so as to maintain them ordered by address. When the buffer cache is full, a non-preemptible kernel thread runs which flushes the cache to the physical device. This thread cooperates with the low level driver of the physical device. With DRAID, the low level driver manages the blocks evicted from the buffer cache by identifying convenient block clusters then acting upon these.

Let us suppose our DRAID be arranged into stripes of $N + K$ disks each. Each block in a block cluster can be encoded by Reed-Solomon as a redundant data stripe formed by $N+K$ segments, $S_1S_2...S_{N+K}$. So, the block cluster can be turned into a cluster of data stripes. By dividing each stripe into its segments then grouping together homologous segments, we turn the cluster of stripes into $N + K$ clusters of segments, each one to be written to a distinct disk. At this point, the DRAID module issues $N + K$ "cumulative" write operations, each directed towards a distinct disk. Each write operation requires an acknowledgement from the destination station/disk, if remote; to allow overlapping latencies among multiple pending write operations, these are implemented with *non-blocking* semantics.

By design, contiguous DRAID virtual blocks result in contiguous data stripes which are kept contiguous on the physical disks. Write clustering thus optimizes both on network communications and on physical disk writes.

2.4.2 Read ahead

Write clustering is only meaningful on write. On read, the Linux buffer cache manager implements a different optimization, namely a *read ahead*. With read ahead, each read operation to a given virtual block B_l actually triggers reading a whole sequence $B_l, B_{l+1}, B_{l+2}, ..., B_{l+r-1}$ of r consecutive blocks, where r is a parameter of the Linux kernel. This is a read of a block cluster, and the low level driver of DRAID accomplishes this by issuing $N + K$ "cumulative" requests, each directed to a distinct station/disk. This way, the number of distinct read requests is independent from the size of the block cluster. Read operations are implemented according to a non-blocking semantics so as to allow overlapping latencies among multiple pending requests.

It must be said that the current prototype of DRAID may suffer from network congestion on read. To read a DRAID virtual block, a given station S must issue $N + K - 1$ read requests to remote stations in a short time interval. Shortly after, as a consequence, S gets a burst of packets coming from many remote disks, which can cause a congestion on the LAN switch. Read ahead leads to even more congestion, because of an increased number of read responses in a short time interval; thus, the parameter r of the read ahead cannot be very large (see Section 3). The communication layer of DRAID is currently based on UDP, with a small extension for packet retransmission but no congestion control. TCP would in principle provide a better solution, being it able to self-adapt to congestions, but would require one socket open towards each remote station/disk, each with a possibly very large window. With large clusters, this would tend to consume all of the available RAM on each host; this is the reason why we preferred not to use TCP as a transport layer for DRAID.

3 DRAID performance

We have measured write and read performance of the bare DRAID virtual block device, with no file system on top of it. The block size of the DRAID virtual block device was set to the maximum allowed value, namely, 4096 bytes. The UNIX command `time dd if=/dev/zero of=/dev/draid count=N` was used to measure the time to write N blocks. Read performance was evaluated by means of command `time dd if=/dev/draid of=/dev/null count=N`. This simple benchmark catches the DRAID behaviour in the case of long sequences of contiguous blocks, which is representative of a practical use of DRAID for reading/writing large files.

The measurement testbed consists of eight single-CPU PCs, each with an AMD Athlon MP 1900+ processor, 512 MB registered ECC RAM, a Maxtor D740X EIDE hard disk at 7200 rpm, and a Netgear GA620 Gigabit Ethernet adapter, all on a GigaByte 7DPXDW motherboard. The PCs are networked by a BATM Titan T5 8-way fiber optics Gigabit Ethernet switch.

In this configuration of DRAID, each PC corresponds to a single disk. Thus, the disk array counts eight disks; six of them are for data, whereas the remanining two ones are for redundancy. We call this configuration a “6+2 DRAID”.

In our measurements, the elapsed time includes the Reed-Solomon encoding, and the total amount of read or written data comprises the redundant part. Thus, a throughput of 100 MByte/s on a “6+2 DRAID” like ours actually means that the user sees $100*6/8$ = 75 MByte/s throughput. We prefer to include the redundant part in our throughput evaluation because we want to normalize w.r.t. the degree of redundancy. Indeed, the more the redundancy, the lower the fraction of throughput delivered to the user level, so a normalized evaluation should rather refer to a “8+0 DRAID” where no redundancy is present. This however would exclude the Reed-Solomon encoder at all, what would lead to over-estimated performance. By using a “6+2” configuration but accounting the redundant part into the total amount of read/written data we get a normalization without excluding the Reed-Solomon encoder. It must be said, however, that the arbitrary choice of a “6+2” configuration has an impact on performance as well, since for instance a “4+4” configuration of the same cluster would lead to a larger fraction of time spent by the Reed-Solomon encoder.

Figure 1 reports the DRAID average write throughput as a function of the total amount of written data, for various degrees of write clustering. Clustering together a lot of write accesses to contiguous blocks appears to be a key feature to attain satisfactory levels of performance. With substantial write clustering, DRAID almost saturates the UDP bandwidth available on Gigabit Ethernet; this is clearly shown in Figure 2, where the asymptotic write throughput (that is, write throughput with a very large data stream) is depicted as a function of the number of blocks per write cluster.

Figure 3 shows the DRAID average read throughput as a function of the total amount of read data,

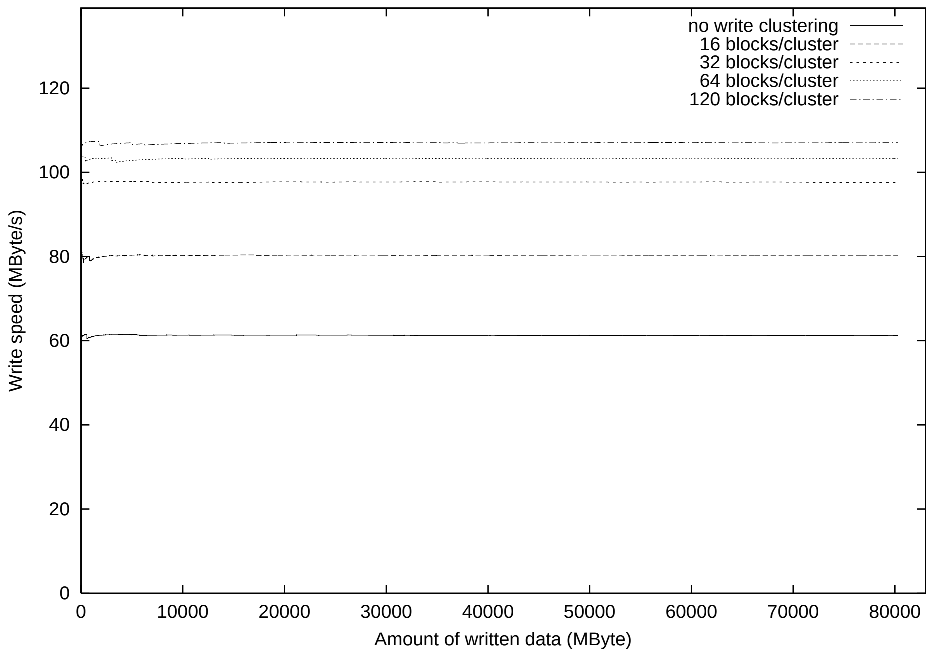

Figure 1: DRAID average write speed as a function of the total amount of written data.

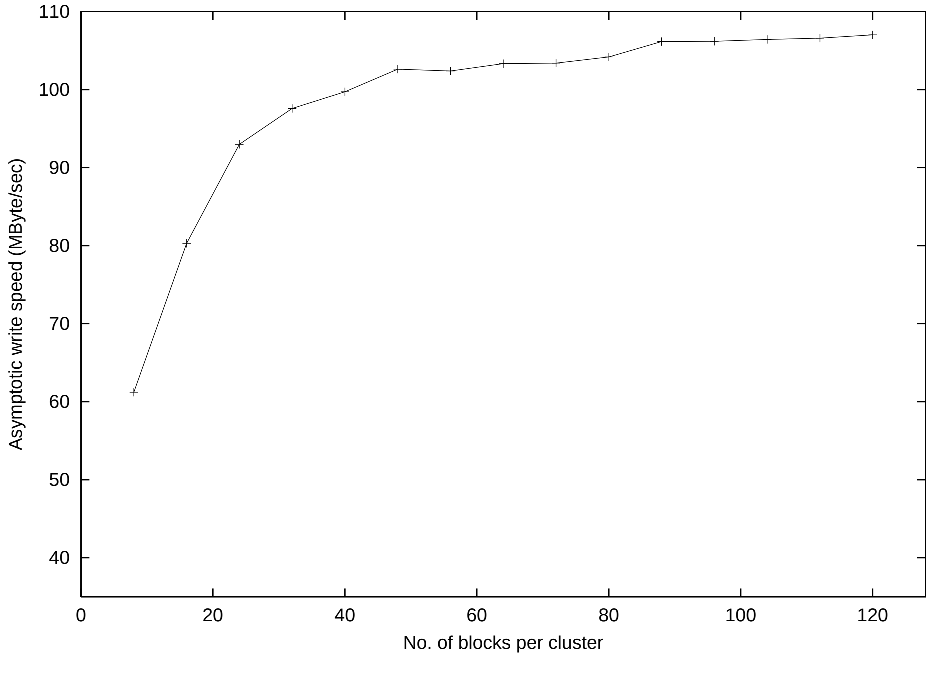

Figure 2: DRAID asymptotic write speed as a function of block cluster size.

for various degrees of read ahead, whereas Figure 4 shows the asymptotic read throughput as a function of the number of blocks requested by a read ahead (including the block actually queried). The positive effects of read ahead are clearly visible, but in this case DRAID could not saturate the network, because

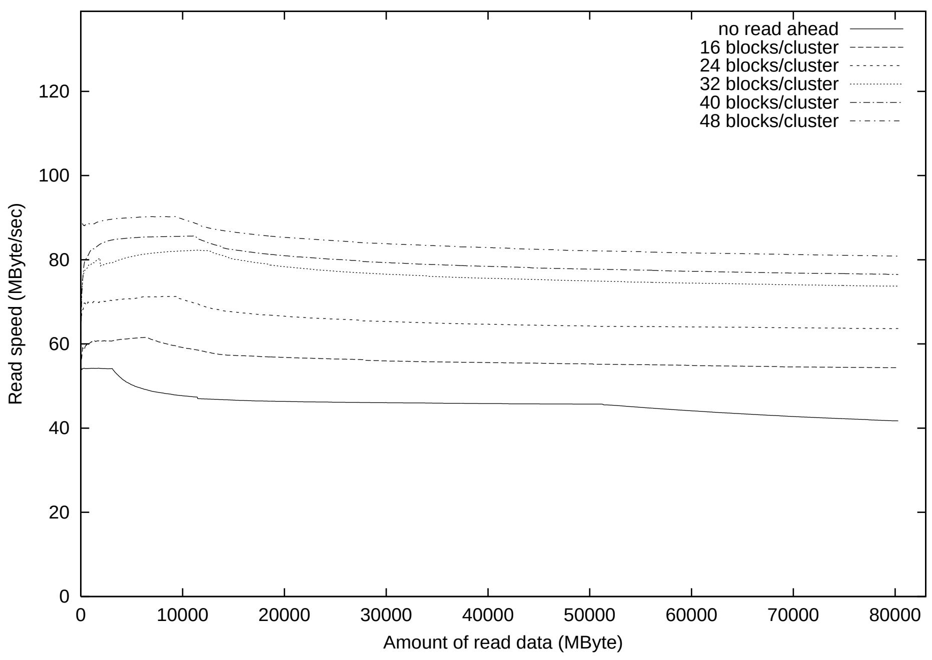

Figure 3: DRAID average read speed as a function of the total amount of read data.

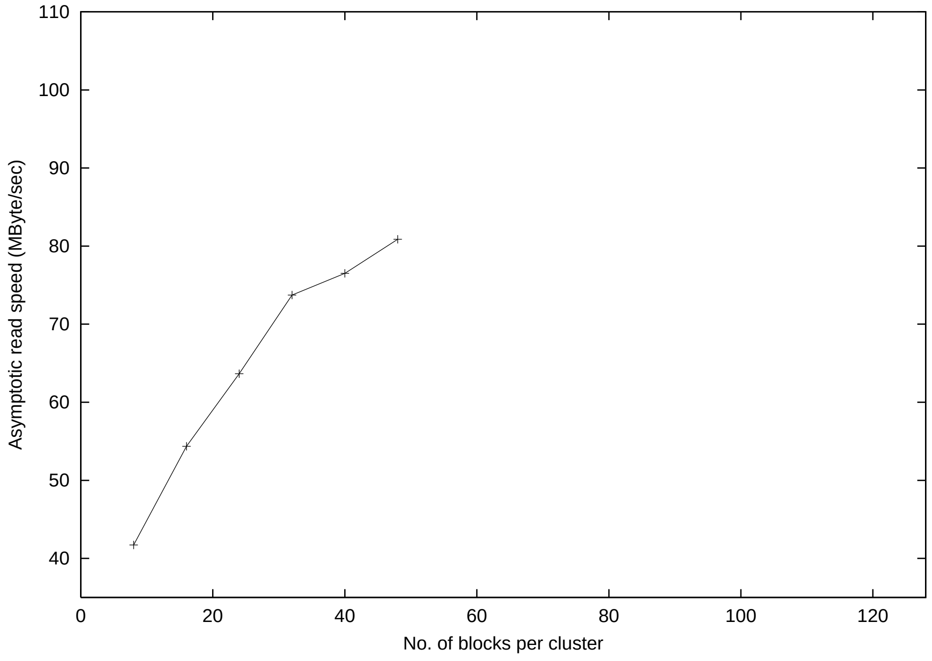

Figure 4: DRAID asymptotic read speed as a function of block cluster size.

when the size of the block clusters increases beyond a certain threshold (48 in our case), the LAN switch becomes congested (Section 2.4.2). When a congestion arises, the switch starts discarding packets. The communication layer of the current prototype of DRAID is capable of retransmitting missing packets, but lacks congestion control (Section 2.4.2). As a result, the systems keeps working but performance drops to very modest levels (in the order of one MByte/s).

4 Related work

Despite the great interest around cluster architectures and storage systems, to the best of our knowledge there are quite few documented research efforts on the practical use of clusters as cost-effective, dependable distributed storage systems. Although the idea dates back to the past decade [15], the performance limits of interconnection networks at that time did not encourage much the practical implementations of distributed RAIDs.

Swift [4] was one of the first prototypes to practically implement the idea of connecting more independent disks in parallel according to a distributed architecture based on a generic interconnection network, as opposed to the centralized, "hard-wired" architecture proposed for traditional RAIDs [10]. As the throughput was of main concern, no redundancy was implemented in that first prototype. Fault tolerance was however added later on [9].

The Petal experiment [8] from the former Digital Equipments Co. proposed a distributed storage system consisting of network-connected storage servers. The project was the first in implementing the concept of a distributed RAID. A Petal system can only tolerate single failures, and provides a single end-point for service access, using a custom RPC-style interface. Petal implements the global address space of the virtual block device at the user level, rather than in the OS kernel; this choice made it necessary to develop a special-purpose file system running at user level as well. A distributed lock functionality is provided, in order to support global lock over a distributed file system. An important contribution of the Petal experiment was indeed the deployment of such a distributed file system, called Frangipani [16]. The Swarm Scalable Storage System [5], based on the GNU/Linux OS, shares many of its basic principles and features with Petal, but provides no data redundancy or distributed lock facilities.

Tertiary disk [3] is a reliable storage system built with a cluster of PCs. Each single node in the cluster comprises two PCs, each with its own SCSI controller, and as many as 14 SCSI disks shared by both PCs. It applies the RAID-5 pattern of data distribution across disks, thus tolerating at most one single disk failure. Duplication of hardware resources (PCs and SCSI controllers) at the node level is meant to improve robustness. A log-based serverless network file system called xFS [1] runs on top of Tertiary Disk.

The RAID-x esperiment [7] is the most recently documented prototype of a distributed RAID we are aware of, and also the first one to implement a SIOS virtual block device across a cluster. Nevertheless, the main focus of RAID-x appears to be on the proposal for yet another RAID mode, called "orthogonal striping and mirroring" by the authors. This RAID mode tolerates at most one disk failure regardless of the total number of workstations involved in the distributed RAID. Tolerating at most one failure clearly does not scale up with the number of nodes, especially when the nodes are workstations operated by humans and as such subject to any kind of physical abuse.

Up to our knowledge, DRAID is the first distributed RAID capable of tolerating multiple failures while providing a SIOS abstraction across a cluster of PC.

5 Conclusions and open points

DRAID is a successful prototype of a distributed RAID implemented on a cluster of PCs. Performance evaluation on a Gigabit Ethernet LAN shows that DRAID is able to saturate the cluster interconnect on write, thus delivering a write throughput in the Gbps range. On read, the current prototype shows a congestion hazard caused by a hot spot on the (switch port connected to the) requestor node in the cluster. The hot spot might be eliminated by implementing a suitable and robust gather algorithm for the data coming from remote disks on read; this is clearly a point open to further investigation.

The encouraging results shown in this paper have been achieved without any specialized piece of hardware or networking driver, and without even modifying the existing operating system kernel. All the software needed to operate the RAID mechanisms is self-contained into a Linux kernel module.

DRAID offers a SIOS abstraction, and is able to tolerate a site-configurable number of disk failures. No other distributed RAID is currently able to provide both such features; yet, SIOS and multiple fault tolerance are of decisive importance for so-called "I/O-centric clusters", namely, clusters of PCs aimed at offering storage services and/or running I/O-intensive tasks. DRAID can be expanded by simple addition of new PCs with local disks, up to saturation of the LAN bandwidth. Larger configurations can be achieved by increasing the throughput of the interconnect; this can be obtained by leveraging next generation LAN hardware, or by exploiting the Linux "channel bonding" networking feature. [2]

The potential of the SIOS block device abstraction for parallel I/O deserves further investigation and evaluation. Another interesting point for investigation is on the effectiveness of DRAID as a support for a shared file system. A promising approach we are going to explore is based on modifying the Linux Virtual File System (VFS) layer, namely, the kernel-level stub to the many Linux file systems, in order to grab information about concurrent accesses to files; this information can be used to minimize the overhead of a distributed algorithm for buffer cache coherency that DRAID still lacks.

[2]The Linux "channel bonding" feature allows one to bind more independent network links to the same virtual network device, provided they all end up on the same hub or remote station. This allows one to communicate to a given partner using more wires in parallel, thus increasing the available throughput.

Acknowledgements

This investigation is supported by the italian National Institute for Nuclear Physics (Istituto Nazionale di Fisica Nucleare, INFN) in the framework of the INFN-GRID research project, and by the italian Ministry for Education, University and Research (Ministero per la Istruzione, l'Università e la Ricerca, MIUR) with a FIRB financial programme.

References

[1] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. In Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP). *ACM Operating Systems Review*, 29(5), 1995.

[2] S. Asami, N. Talagala, T. Anderson, K. Lutz, and D. Patterson. The Design of Large-Scale, Do-It-Yourself RAIDs. *http://www.cs.berkeley.edu/ pattrsn/papers.html*, November 1995.

[3] S. Asami, N. Talagala, and D. Patterson. Designing a Self-Maintaining Storage System. In *16th IEEE Symp. on Mass Storage Systems*, San Diego, CA, March 1999. IEEE.

[4] L. F. Cabrera and D. D. E. Long. Swift: a storage architecture for large objects. In *11th IEEE Symp. on Mass Storage Systems*, pages 123–128. IEEE, October 1991.

[5] J. H. Hartman, I. Murdock, and T. Spalink. The Swarm Scalable Storage System. In *Proc. of the 19th Int'l Conf. on Distributed Computing Systems (ICDCS)*, pages 74–81, Austin, TX, May 1999. IEEE.

[6] K. Hwang, H. Jin, E. Chow, C. Wang, and Z. Xu. Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space. *IEEE Concurrency*, 7(1):60–69, 1999.

[7] Roy S. C. Ho Kai Hwang, Hai Jin. Orthogonal Striping and Mirroring in Distributed RAID for I/O-Centric Cluster Computing. *IEEE Trans. on Parallel and Distributed Systems*, 13(1):26–44, 2002.

[8] E. K. Lee and C. A. Thekkath. Petal: Distributed Virtual Disks. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Cambridge, MA, October 1996. ACM Press.

[9] D. D. E. Long and B. R. Montague. Swift/RAID: A Distributed RAID System. *Computing Systems*, 7(3):333–359, 1994.

[10] D. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of ACM SIGMOD Annual Conference*, pages 109–116, Chicago, IL, June 1988. ACM Press.

[11] W. W. Peterson and Jr. E. J. Weldon. *Error-Correcting Codes, 2nd ed.* MIT Press, Cambridge, MA, 1972.

[12] G. F. Pfister. The Varieties of Single System Image. In *Proc. of IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 59–63. IEEE, 1993.

[13] A. L. N. Reddy and P. Banerjee. An Evaluation of Multiple-Disk I/O Systems. *IEEE Trans. on Computers*, 38(12):1680–1690, December 1989.

[14] K. Salem and H. Garcia Molina. Disk Striping. In *Proceedings of the Second International Conference on Data Engineering*, pages 336–342, Los Angeles, CA, February 1986. IEEE Computer Society.

[15] M. Stonebraker and G. A. Schloss. Distributed RAID - A New Multiple Copy Algorithm. In *Proc. of the 6th Int'l Conf. on Data Engineering*, pages 430–437, Los Angeles, CA, February 1990. IEEE.

[16] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP)*, Saint Malo, France, October 1997. ACM Press.