An Implementation of a Highly Reliable Parallel-Disk System using Transputers

Seishi Tomonaga Haruo Yokota

School of Information Science
Japan Advanced Institute of Science and Technology, Hokuriku
15 Asahidai, Tatsunokuchi, Nomi, Ishikawa, 923–12 JAPAN
E-mail: tomo@jaist.ac.jp, yokota@jaist.ac.jp

Abstract

There are many approaches for improving the performance and reliability of secondary storage. The RAID (Redundant Arrays of Inexpensive Disks) is an example, but it still has a reduced performance for reconstructing data and using a large number of disks due to a bottleneck in the bus.

We proposed DR-nets (Data-Reconstruction networks) to remove this bottleneck. In DR-nets, each node is connected by an interconnection network instead of the bus. Due to this interconnection network, the communication among nodes keeps locality in reconstructing data and the scalability can be increased. DR-net can also reconstruct data using parity information.

We implemented an experimental system using Transputers. It contains 26 inmos T805s and 25 Quantum hard disk drives. The control software on each node is written in Occam. The processes for routing packets and controlling disk accesses are executed in parallel.

We describe the hardware and software configurations of the system, and report some experimental results.

Keywords — disk arrays, RAID, interconnection networks, reliability, file allocation, Occam

1 Introduction

There is a strong demand for high performance and reliability in computer systems. Secondary storage may prevent from reaching these demands. There are some problems of lowering system performance by secondary storage, known as Amdahl's Law, and of low-reliability caused by mechanical movements on disks. Solving these problems are essential to the growth of computer systems.

A Redundant Arrays of Inexpensive Disks (RAID) was proposed by D.A.Patterson et al. of UC Berkeley [PGK88] to solve the above-mentioned problems using data-striping and redundant information. It stores data and redundant information into multiple disks, and it is classified into five levels corresponding to the method of storing the redundant information:

level 1 Mirrored disks. It duplicates all data as redundant information.

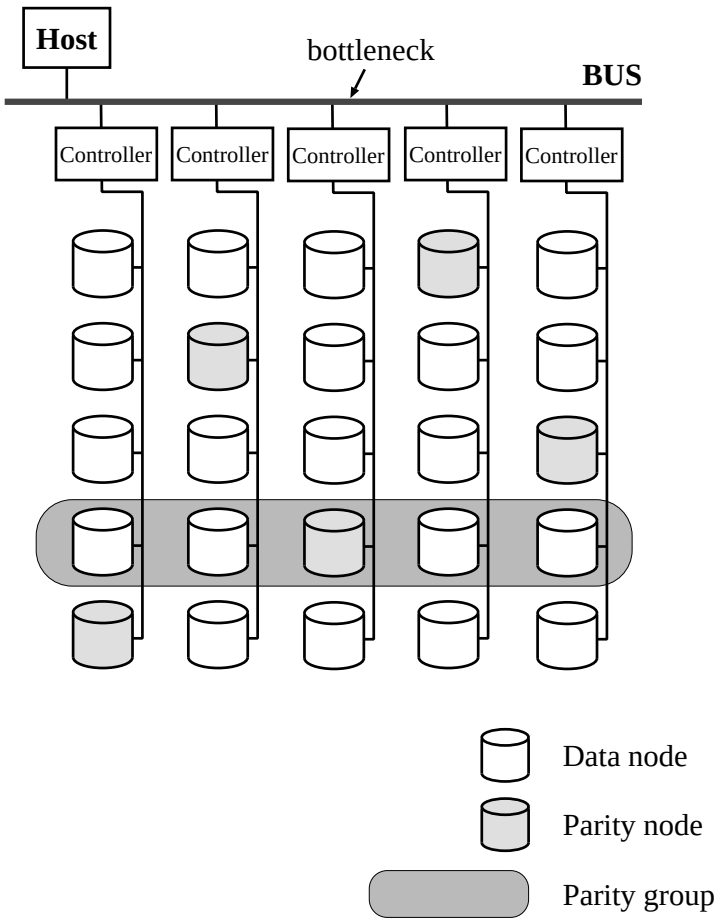

Figure 1: An Ordinary RAID System

level 2 It stores disks with Hamming coded data in bits.

level 3 It uses parity codes instead of Hamming codes for correcting errors since disk controllers can detect damaged disks. Data is also interleaved in bits.

level 4 Data is interleaved in sectors instead of bits. It makes each disk access independent. Parity information is stored in a single disk in a group.

level 5 It spreads parity information into all disks in a group.

Figure 1 shows an ordinary level 5 RAID system.
In level 4 and 5 RAID, we can calculate new parity codes as follows [PGK88]:

$$new\ parity = (old\ data\ \mathbf{xor}\ new\ data)\ \mathbf{xor}\ old\ parity \qquad (1)$$

Two disks, a data disk and the parity disk, are accessed to modify the parity codes.

RAID needs to calculate the expression (1) in every access of write data. Also it is necessary to apply the **xor** among the parity codes and data in a group to reconstruct the data under failure. Thus, the bus becomes the bottleneck because all disks in the group are accessed for reconstruction.

In Section 2, we first give an outline of the DR-net which can alleviate the bus bottleneck by using an interconnection network and improve the reliability of the system.

We introduce an experimental system for the DR-net in Section 3. We describe the hardware configuration and then the software implementation. Section 4 reports some results.

2 The DR-net

2.1 Applying RAID on an Interconnection Network

RAID has been proposed for improving access speed and reliability of secondary storage. As mentioned above, however, performance is reduced due to the bus. The clustered RAID [ML90][MY92] or the RAID system connected by a network [LCH+92] have been proposed, but both are not an optimized to this bottleneck.

We propose *DR-nets* (Data-Reconstruction networks) to solve these problems [Yok93b] [Yok93a]. In the DR-net, disks are connected to each node of an interconnection network and data is scattered with parity information to each node.

A parity group is composed of neighboring nodes interconnected around a central node. For example, the parity group is composed of five nodes in the shape of a *cross* if we adopt a two-dimensional torus network as network topology. It is possible to make five parity groups on the 5×5 torus network (Figure 2). If we have non-overlapped parity groups like Figure 2, there will be no collision on the links for the case of modifying parity codes and/or reconstructing data. As such, we are able to resolve the bus bottleneck using an interconnection network and keep locality to access data.

We will go into details on the operation of the nodes in Section 3.2.3.

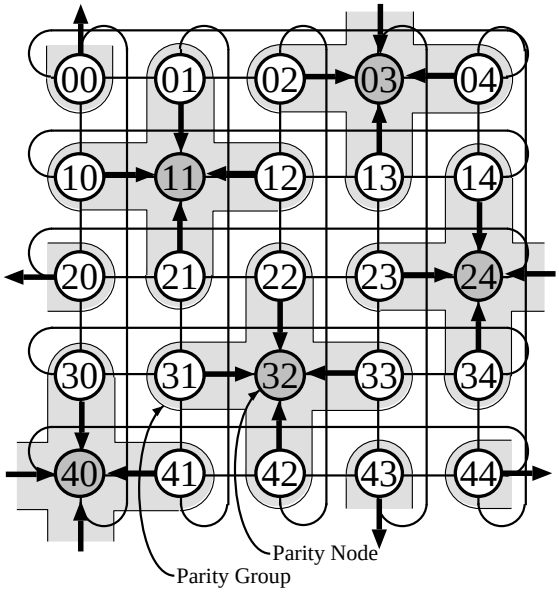

Figure 2: First Parity Groups

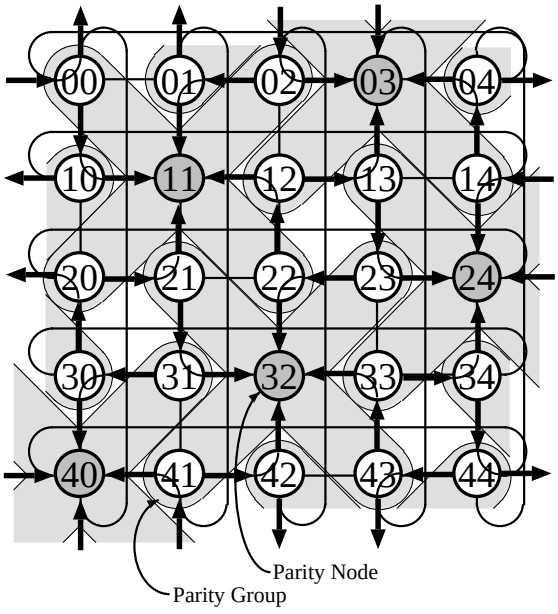

Figure 3: Second Parity Groups

2.2 Improving Reliability

To a certain extent, we can achieve higher reliability using an interconnection network and non-overlapped parity groups. A simple calculation shows that the system can repair all data in the presence of on average 3.9 faults for 25 disks without repair [Yok93b][Yok93a]. It has the same reliability as for RAID, but it is just an average measure, and data can be lost by only two faults in the same group under the worst case conditions.

Now we consider improving system reliability to allow multiple faults. We have to introduce an increase in redundancy for improving reliability. The system performance will be reduced by this additional redundancy, but care must be taken not to allow overlapping of parity groups and/or link collisions. Here we adopt a method of constructing second parity groups (*SPGs*) on the same interconnection network of first parity groups (*FPGs*) described in the previous section. Figure 3 illustrates the SPGs that have no overlapping and no link collision among parity groups on the 5×5 torus network. The communication among nodes forms *swastikas* or *reverse swastikas*. Figure 4 illustrates an example of recovering faults with FPGs and SPGs. Initially, two SPGs reconstruct data in *node(22)* and *node(33)*, then FPG reconstructs data in *node(31)*.

It becomes possible to recover data by introducing SPGs even if any two disks are damaged in a system using 25 nodes [Yok93c][Yok93a], data is recovered 95% for four damaged disks.

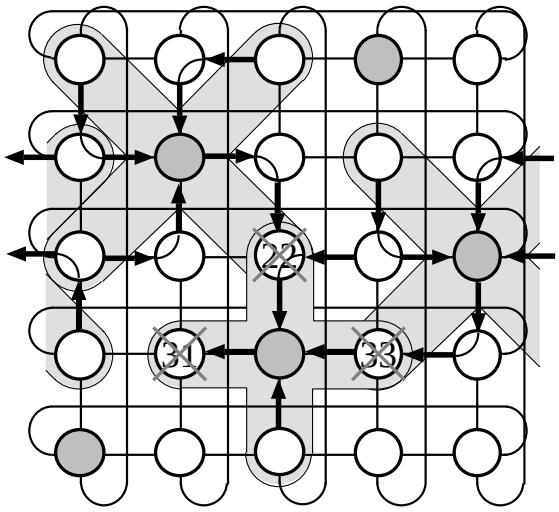

Figure 4: Recovering Faults with FPGs and SPGs

3 An Experimental System

In this section, we describe an experimental system of DR-net using Transputers. First, we describe briefly the hardware configuration, then explain the software implementation.

3.1 Hardware Overview

The experimental system is shaped as a tower including five VME subracks (Figure 5). There are also five plug-in-units in each VME subrack. The plug-in-unit consists of the following components:

- A CORAL HPT04 SCSI-2 TRAM (SCSI controller) using an inmos T805 (25MHz)
- An IMS B014 VMEbus board with C004 link switch chips
- A Quantum Go·Drive 120S (2.5" 120MB SCSI hard disk drive)
- A power supply
- A case

In front of each plug-in-unit, there are a disk access indicator and a power switch of a disk (which simulates the disk failure). Since each plug-in-unit is removable, we can easily replace any damaged disks.

The Quantum hard disk drive supports an average seek of 17ms, an average rotational latency of 8.3ms, and a maximum transfer rate of 4MB/s.

Each T805 is connected to neighboring nodes by links. It is impossible to make the 5×5 torus system communicate to the outside only by 25 T805s because a T805 has only four hardware links. We use an extra T805 besides SCSI TRAM as a root node. The root node is used for host communication. Thus, we use 26 Transputers to form the 5×5 torus network. We use the system via Ethernet using an IMS B300 TCPlink hardware. Figure 6 illustrates the network construction of the experimental system.

Since we use only one root Transputer for host communication, the traffic toward/from the root node may be heavier. We can add more Transputers for host communication into the torus connections to alleviate this bottleneck.

The specification of communication bandwidth among nodes is 20Mbits/s. We measured it with an IMS C004 link switch chips on the VMEbus boards and we obtained 1.1MBytes/s (8.8Mbits/s) for one direction.

3.2 Software Configuration

In the DR-net, a file is divided into some *fragments*, then sent to each node as *packets* and stored. In this section, we describe the generation of the packets, the construction of routing/executing processes, the behavior of the executing process under/without failure, and a method for storing data. All programs are written in Occam.

3.2.1 Protocol among Nodes

Each node has its own address in the interconnection network. Packets are transmitted from a source node to a destination node using the address. The packet datagram delivered among nodes is as follows:

source addr	relay addr	destination addr	fragment #	operation	page # in disk	data

The address of the source and destination nodes is used for sending and/or receiving (i.e. sending back). The address of a relay node is used for avoiding link collisions. By

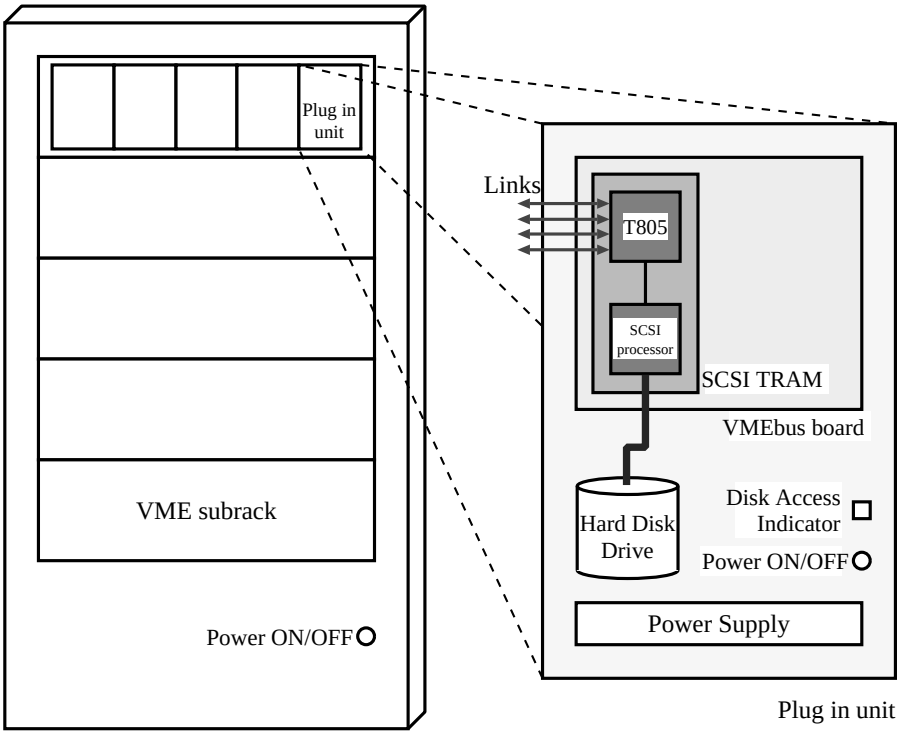

Figure 5: Hardware Configuration of the Experimental System

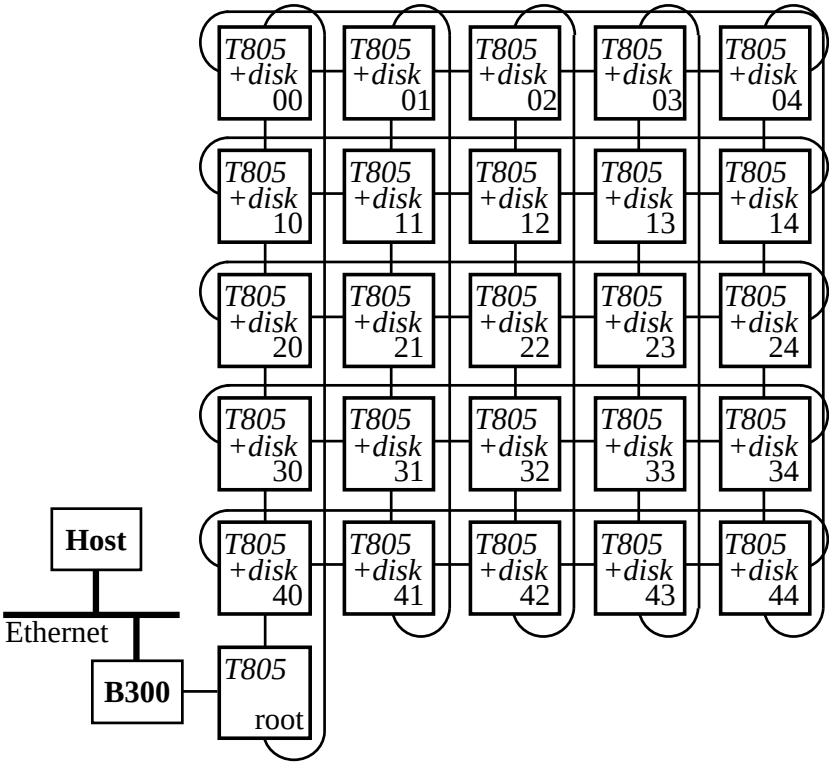

Figure 6: Network Construction of the Experimental System

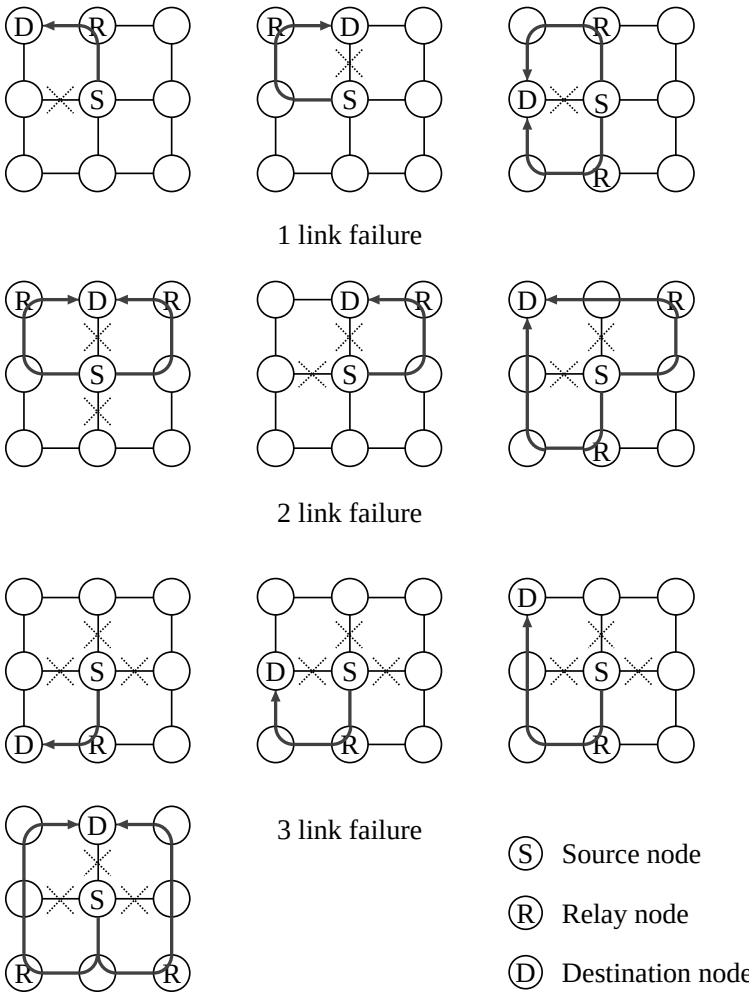

Figure 7: An Example of Avoiding Link Failure

specifying the relay node, this makes it possible to deliver packets with no link collision when the packets are transferred in the SPGs. The path from the source node to the relay node or from the relay node to the destination node is optimized.

It is, moreover, possible to avoid link failure using a dynamic routing method which dynamically re-specifies the relay node. Figure 7 illustrates an example of avoiding link failures. The S, R and D in this figure correspond to a source node, a relay node and a destination node respectively. A packet can avoid some link failures using the relay node when it is sent to the eight neighboring nodes.

The field of the fragment number in the packet is used for reassembling the file. The file allocation policy is described in Section 3.2.4, and its operation in Section 3.2.3.

3.2.2 Processes and Channels

Each node must deliver packets to their destination nodes independent of the assigned tasks. Figure 8 illustrates the processes and the channels in each node. Four senders/receivers to/from each direction and executors run in parallel.

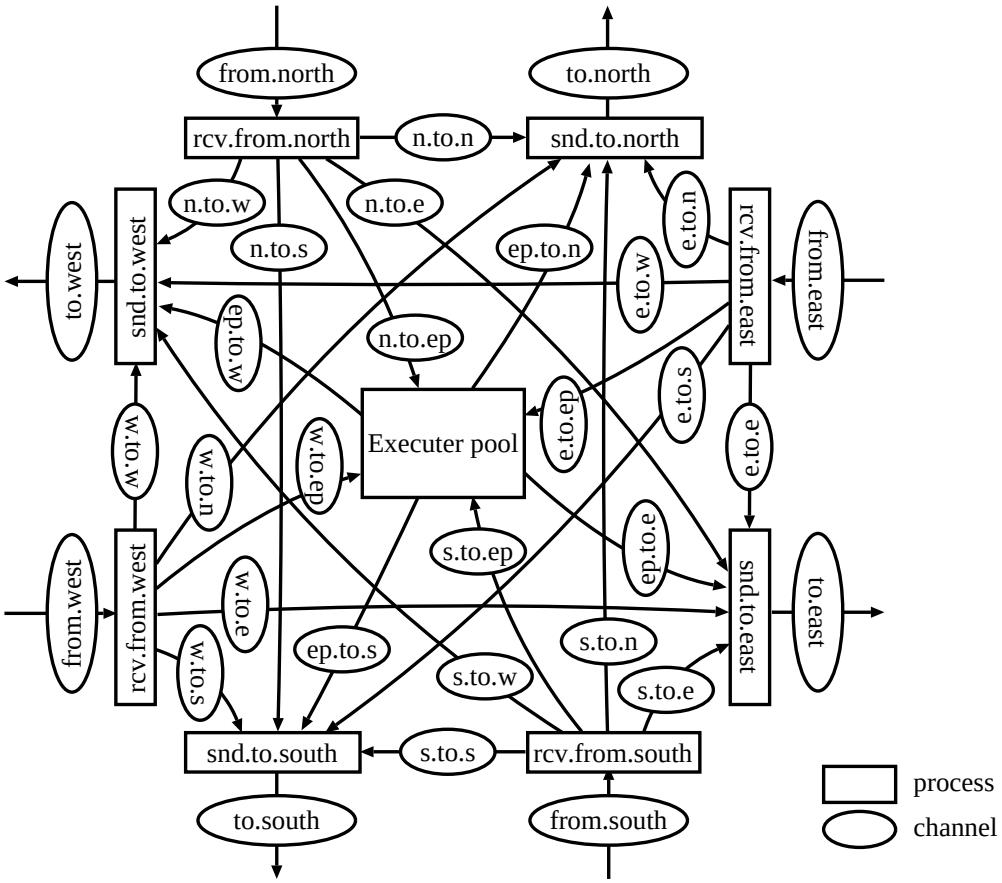

Figure 8: Processes and Channels

The Occam program in each node is outlined below:

```
...  declare procedures
...  declare channels
PAR
  -- senders
  snd.to.north(to.north, n.to.n, s.to.n, e.to.n, w.to.n, ep.to.n)
  snd.to.south(to.south, n.to.s, s.to.s, e.to.s, w.to.s, ep.to.s)
  snd.to.east(  to.east, n.to.e, s.to.e, e.to.e, w.to.e, ep.to.e)
  snd.to.west(  to.west, n.to.w, s.to.w, e.to.w, w.to.w, ep.to.w)

  -- receivers
  rcv.from.north(from.north, n.to.n, n.to.s, n.to.e, n.to.w, n.to.ep)
  rcv.from.south(from.south, s.to.n, s.to.s, s.to.e, s.to.w, s.to.ep)
  rcv.from.east(  from.east, e.to.n, e.to.s, e.to.e, e.to.w, e.to.ep)
  rcv.from.west(  from.west, w.to.n, w.to.s, w.to.e, w.to.w, w.to.ep)

  -- executors
  executor.pool(ep.to.n, ep.to.s, ep.to.e, ep.to.w,
                n.to.ep, s.to.ep, e.to.ep, w.to.ep)
```

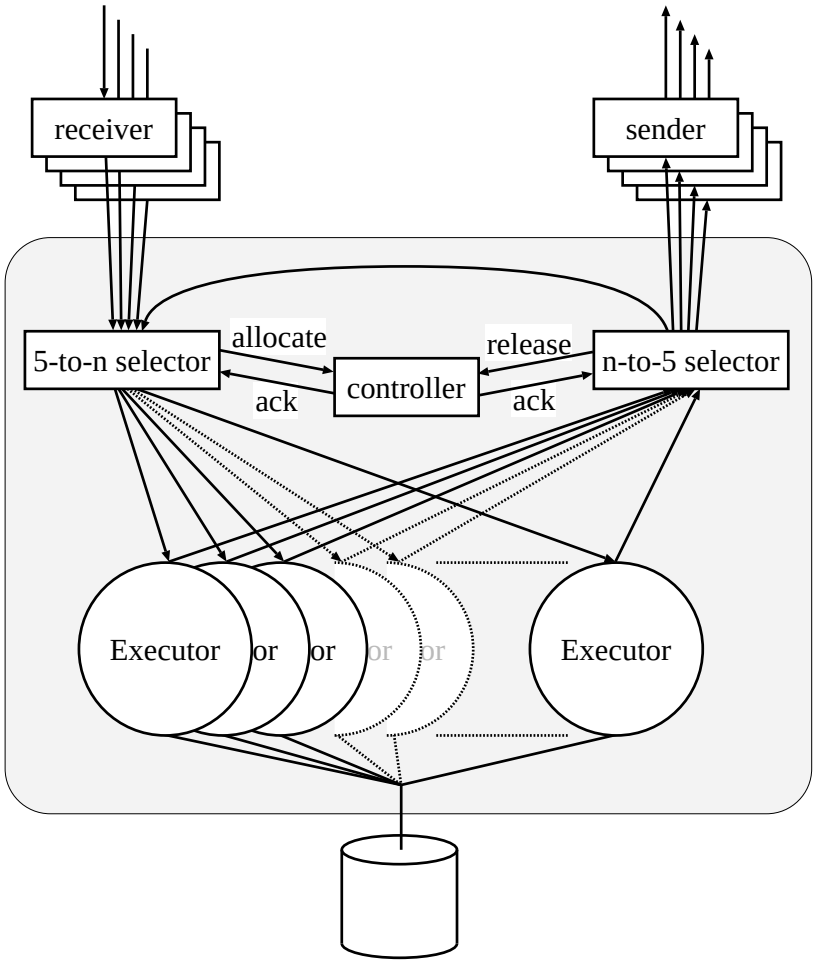

Figure 9: Inside of the Executor Pool

At first, a receiver receives a packet and checks the address of the destination node. If the destination address is not same as its own address, then a sender sends the packet so that it follows an optimum route to give priority to the horizontal direction through a channel between the receiver and the sender.

If the destination is equal to its own address, then the packet is sent to the executor pool. In the executor pool, the packet is allocated to an executor by the controller and sent to the executor. The executor allocated to the packet reads or writes data from/to disk, and sends the result to the appropriate node via a sender (Figure 9).

3.2.3 Operations in Executors

There are five types of operations in executors which correspond to the behaviors of each node. Figure 10 illustrates the behaviors and movements of packets.

a. Read without disk failure An executor reads a page from the disk and sends back a packet to an appropriate node given by the source address field of the request packet.

After sending it back, the executor is released.

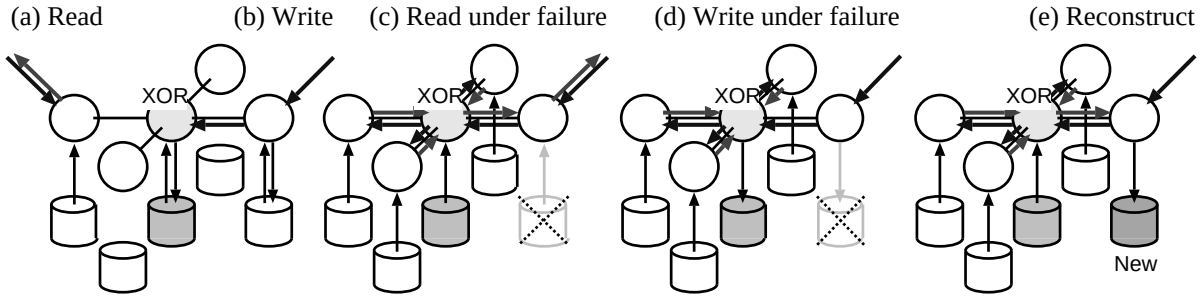

Figure 10: Behaviors of Nodes and Packets

b. Write without disk failure A write operation contains four disk accesses.

1. An executor reads the old data from the disk.
2. The executor writes the new data to the disk.
   And the executor simultaneously applies **xor** to the old and new data, then sends the **xor**-ed data to the parity node.
3. An executor which receives the packet in the parity node reads the old parity data from the parity disk.
4. The executor in the parity node applies the **xor** to the received data and old parity data, then it writes the new parity data to the parity disk.

The executor in the object node is in charge of the first two steps, while the executor in the parity node is in charge of the last two steps. The write operation in Step 2 and the read operation in Step 3 are executed in parallel.

After sending the packet to the parity node, the executor in the object node is released.

c. Read under disk failure When an object disk is damaged and is impossible to read data from the disk, then the executors behave as follows:

1. An executor in the object node sends a read request to the parity node.
2. An executor which receives the request in the parity node reads the parity data from the parity disk.
   The executor in the parity node sends read requests to the remaining nodes of the parity group.
3. The executor in the parity node applies **xor** to the parity data and packets as soon as it receives from the remaining nodes.
   After applying **xor** to all packets from the three remaining nodes, the executor in the parity node sends back the result to the object node.
4. The executor in the object node sends back the packet to the node which issued the read request.

The executor in the object node is in charge of Steps 1 and 4, and the executor in the parity node is in charge of Steps 2 and 3. After executing its normal processing, the executors in each node are released.

d. Write under disk failure When an object disk is damaged and it is impossible to write data to the disk, then the executor in the object node issues a write request to the parity node. The executor receiving the request in the parity node sends a read request to the remaining nodes of the parity group. Also, the executor in the parity node applies **xor** to the packets from the remaining nodes in succession.

After applying **xor** to all packets, the executor in the parity node writes the new parity data to the parity disk.

The executors in each node are then released.

e. Data reconstruction for a new replaced disk This operation is similar to (c.). An executor in an object node writes the reconstructed data to a new replaced disk instead of sending it back to the node which issued the read request.

We assume that there is no intermittent disk failure. If an intermittent failure occur, and an executor does not detect or check it, then the consistency of data will be lost. In other words, the data is lost if an executor accesses data which must be reconstructed after replacing a disk. This is the reason why a disk once damaged may have inconsistent data.

3.2.4 File allocation

We now propose a method for storing file allocation tables. It is useful if the file allocation tables are scattered and multiplexed as there is no hot spot of access nor the impossibility of access due to loss of tables by disk failures. The following shows how to store files:

1. The root node divides a file into some fragments.

   The node generates a table containing the file name, file size, date and time, and the number of the fragments.

2. The root node sends the fragments to a parity group as packets.

   By round robin it is determined to which parity group the packets are sent.

3. The parity group receiving the packets stores it into the disk of the data nodes in arrival order using a round robin.

   The node generates a table containing the location of storing data, the location of a node and the page number in the disk, and informs the root node of the location of this table. The root node stores this data with the above information.

4. The parity group sends the remaining data to another parity group when the group finished storing around to all data nodes in the group.

5. Another parity group which receives the remaining data also stores them to the data nodes in round robin in arrival order. Also it informs the location of the table to the source parity group.

   The source parity group also stores this data with the above information.

6. The steps mentioned above are repeated till all of the packets are stored.

4 Access Speed

Figure 11 illustrates the time table for writing a page and modifying the parity. For the measurements using the experimental system, random access to maximize seek time and an average of ten measurements were specified. 512bytes is used as an access size of a page.

The time for reading a page is 25ms, while writing takes 6ms. Writing data and modifying parity take 57ms. The 6ms, for writing, is not an actual time for writing a page to a disk, but the time for writing it to a buffer of a disk controller. As at present, we now do not have a good way of measuring the absolute time for writing.

We have not measured the time for the **xor** or send/receive because they are very small. Incidentally, the transfer rate of the send/receive process is 1.1MBytes/sec and the time to set it up is about 3μsec. For transferring a page of data, for example, it takes about 0.5ms.

The time for writing a page and modifying parity is about three times (25ms vs. 57ms) as much as for writing a page provided it takes as much time as for reading a page. This result is correct because writing in the data node and reading in the parity node must be overlapped. We can overlap both read and write operations of data and parity. However, since another synchronization mechanism between executors in both nodes is required, we have not adopted this method yet.

The time tables for reading and writing a page under the presence of a failure are shown in Figure 12 and 13. The time for detecting disk errors is 237ms, 274ms for the reading process and 284ms for the writing process. Since errors are detected by a device timeout in our experiment, it can take a lot of time. The real time for reconstructing data except detecting errors is 37ms and 47ms for reading and writing, respectively. We take these results to be correct too.

Data using both FPGs and SPGs has not been gathered, but we expect that these results are the same as using FPGs only. Data transfer takes less time compared with a disk access, and each access to the disk in the FPGs and SPGs are executed in parallel.

5 Concluding Remarks

We have described the implementation and some results for an experimental system using the DR-net. DR-net was proposed to improve the performance and reliability of secondary storage based on RAID. DR-net can reconstruct data using local communication of parity groups in sub-networks of an interconnection network. DR-net has two types of parity groups, FPGs and SPGs. It can achieve higher reliability by increasing the level of redundancy.

We have developed an experimental system using Transputers and its control software in Occam. We have adopted a method for file fragmentation and a composition of packets for transmission among nodes, and detailed them in this paper. The packets with not only destination but relay addresses are delivered with no collision among nodes.

In each node, the routing and the disk controlling processes are executed in parallel. A node can accept multiple requests. Also we have proposed a method for storing file allocation tables to retain the information under disk failures.

We have completed the implementation of the FPGs, and measured the behavior of disk accesses and the access time to the disk of the system. We are now implementing

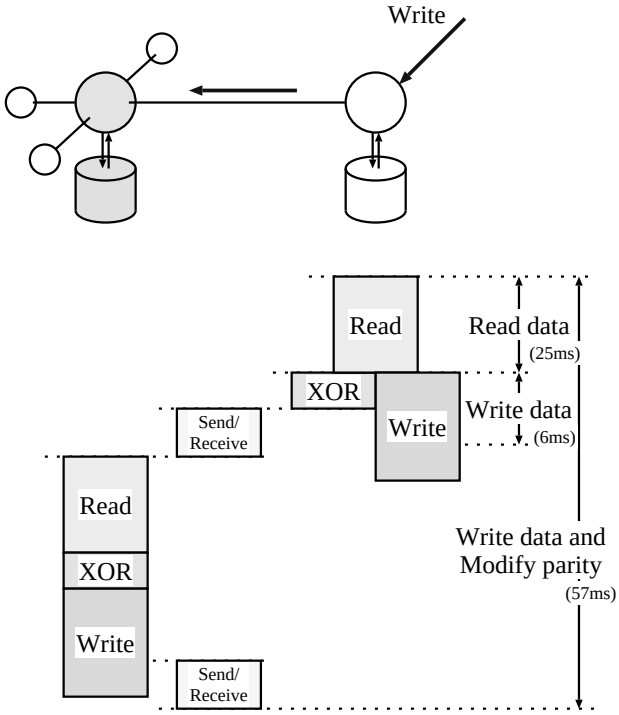

Figure 11: A Time Table for Writing a Page and Modifying Parity

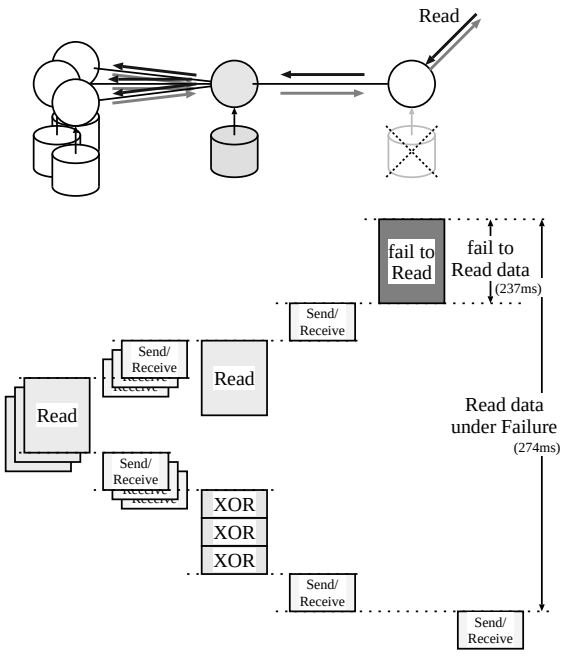

Figure 12: A Time Table for Reading a Page under Failure

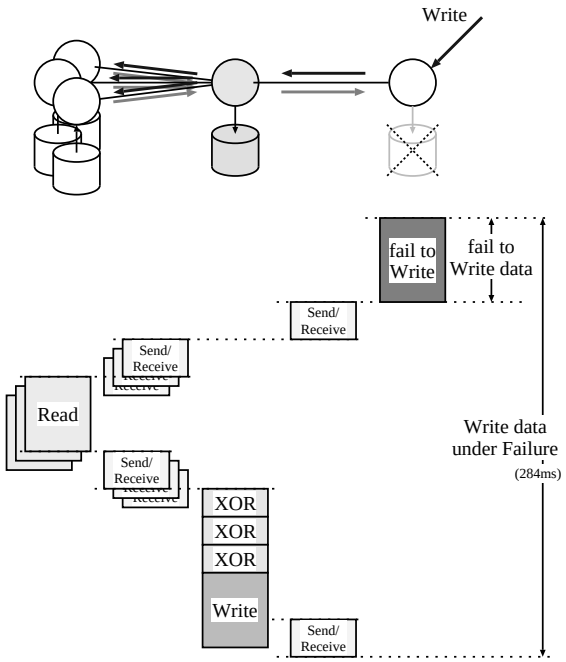

Figure 13: A Time Table for Writing a Page under Failure

the SPGs and the file system, and plan to examine the performance of the DR-net under multiple disk access requests. The performance under many accesses to the disks and/or multiple faults in disks will be investigated.

We are planning to implement the DR-net for other topologies such as the three-dimensional torus or hypercube. We will also consider parallel database operations on the DR-net.

Acknowledgements

We thank Dr.Fabrizio Lombardi of Texas A&M University for his helpful advice on refining this paper.

References

[COR] CORAL. *HPT04 SCSI-2 TRAM User Manual.*

[Gib89] Garth A. Gibson. Performance and Reliability in Redundant Arrays of Inexpensive Disks. In *Proc. of 1989 CMG Annual Conference*, 1989.

[inm] inmos. *IMS B014 User Manual and Reference Guide.*

[LCH+92] Edward K. Lee, Peter M. Chen, John H. Hartman, Ann L. Chervenak Drapeau, Ethan L. Miller, Randy H. Katz, Garth A. Gibson, and David A. Patterson. RAID-II: A Scalable Storage Architecture for High-Bandwidth Network File Service. Technical Report of UCB(EECS) 92/672, UCB/CSD, 1992.

[ML90] Richard R. Muntz and John C.S. Lui. Performance Analysis of Disk Arrays Under Failure. In *Proc. of 16th VLDB Conference*, Aug 1990.

[MY92] A. Merchat and P. S. Yu. Design and Modeling of Clustered RAID. In *Proc. of 22nd FTCS*, 1992.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks(RAID). In *Proc. of SIGMOD 1988*, pages 109–116. ACM, Jun 1988.

[Qua] Quantum. *Go·Drive 60/120S Hard Disk Drive Product Manual.*

[Yok93a] Haruo Yokota. DR-nets: Data-Reconstruction Networks for Highly Reliable Parallel-Disk Systems. Research Report of JAIST IS-RR-93-0010A, Japan Advanced Institute of Science and Technology, Sep 1993. (Submitted for publication).

[Yok93b] Haruo Yokota. On Applying RAID to Networks and Improving Reliability (in Japanese). CPSY 93–11, IEICE, Apr 1993.

[Yok93c] Haruo Yokota. Treatment for Imbalance of Data Reconstruct Networks (in Japanese). FTS 93–20, IEICE, Aug 1993.