This is Chapter 9 from the second edition of :

Networks, Routers and Transputers:

Function, Performance and applications

Edited by: M.D. May, P.W. Thompson, and P.H. Welch

IOS Press
Van Diemenstraat 94
1013 CN Amsterdam
Netherlands

IOS Press, Inc.
P.O. Box 10558
Burke, VA 22009–0558
U.S.A.

IOS Press/Lavis Marketing
73 Lime Walk
Headington
Oxford OX3 7AD
England

Kaigai Publications, Ltd.
21 Kanda Tsukasa–Cho 2–Chome
Chiyoda–Ka
Tokyo 101
Japan

This chapter was written by J.M. Kerridge.

9 The Implementation of Large Parallel Database Machines on T9000 and C104 Networks

The design of large database machines requires the resulting implementation be scalable and cheap. This means that use has to be made of commodity items whenever possible. The design also has to ensure that scalability is incorporated into the machine from its inception rather than as an after–thought. Scalability manifests itself in two different ways. First, the initial size of a system when it is installed should be determined by the performance and size requirements of the desired application at that time. Secondly, the system should be scalable as processing requirements change during the life–time of the system. The T9000 and C104 provide a means of designing a large parallel database machine which can be constructed from commodity components in a manner that permits easy scalability.

9.1 Database Machines

A database machine provides a high level interface to the stored data so that the user is not aware of the access path to that data. Further, the user can specify what data is required and not how the data is to be found. In a relational database machine, the topic of this paper, the data is stored in tables. Each row of a table contains a number of columns each of which contain a single atomic value. Rows are distinguished from each other by the value of one of the columns having a distinct value. Data from one table can be combined with that from another by a process known as relational join. If we assume that in each table there is a column which holds data from the same domain, then we can join the tables on those columns. In general, the output from a join is the concatenation of one row from each of the tables where the joining columns have equal values.

A database machine allows different users to access the database at the same time for any operation. Thus different users can be accessing the database to read, write, modify and erase rows of tables. The effect of each user has to be made invisible to the other users until a user has indicated that a unit of work is complete. The database machine therefore has to ensure that different users do not interfere with each other by accessing the same rows of a table. Many users can access the same row of a table provided they are all reading the data. The maintenance of such a concurrency management system is expensive and most of the current algorithms are based upon the use of a large memory to hold locking information. The design to be presented in this paper will show how a scalable concurrency management system can be constructed.

It is vital that the data stored in the database is correct and consistent. This means that data values have to be checked whenever data is written, erased and modified. This consistency is achieved by the use of integrity constraints which can be of several different kinds. First, there is a simple check constraint to ensure that a value is contained within a simple range of values. A second more complex check constraint can be invoked which ensures that a value in a column of a table is related in some way to a value in another row of the same table, or on some function applied to the table as a whole. This can then be extended to a check which refers to another table. Finally, a referential constraint imposes relationships between tables. The column, or columns, which uniquely identify a row in a table are called the PRIMARY KEY of that table. Another table may store the same values in a column of that table. This column will not be the primary key of the second table, though it may form part of the primary key of the second table. The database system has to ensure that only values which occur in the first table are stored in the second table. The column in the second table is said to be a FOREIGN KEY which references the first table. If we insert a row into the second table then we must check that the value of the foreign key (or keys) occurring in that row already exists in the referenced table (or tables). Similarly, if a row is to be deleted from the first table then we must ensure that there are no rows in the second table which have the foreign key column with the same value as that which is to be deleted. In either case, if this referential constraint fails then the operation on the database should be terminated. It is generally agreed that if full constraint checking is imposed on existing database implementations then the performance of the system will be reduced to 25% of current performance. Thus many database systems are run without consistency checking, especially referential checking, so that the overhead is not imposed. The design to be discussed in this paper will permit the implementation of a full constraint system with a scalable performance.

A key aspect of current database technology is the ability to manipulate complex data types. This is manifested in the interest in object oriented databases. We shall describe how object oriented capabilities are captured by the design.

A final factor which is crucial to database machine performance is that of recovery from errors. Oates and Kerridge [1][2] have shown how a recovery system can be implemented in parallel with the data manipulation component of a database machine. The architecture to be described in this paper will show how these capabilities can be captured.

Many of the ideas expressed in this paper result from the highly successful IDIOMS [3][4]project which resulted in the demonstration of a database machine which could undertake both On–line Transaction Processing (OLTP) and Management Information System (MIS) queries on the same data concurrently. The IDIOMS machine demonstrated this capability for banking applications specified by the Trustees Savings Bank plc. One purpose of this demonstrator was to show that a low–cost scalable architecture could be constructed. This aspect is further enhanced with the use of T9000 and C104 technology.

9.2 Review of the T8 Design

In this section a brief overview of the IDIOMS design is presented. It demonstrates the limitations of the T8 transputer as a basis for building a system which can be scaled easily. Scalability manifests itself in two different ways. First, a system has to be scaled to match the initial size of the application, thereby dealing with different sized applications. Subsequently, the system has to be scaled to deal with changes of application. For example, the amount of data or the number of applications may increase or the response time of the system may have to be improved. Figure 9.1 shows the basic IDIOMS design. Transactions are passed to the T processors, where access is made to the disc for the required records to undertake the transaction. It is presumed that data is partitioned over the discs connected to the T processors. In this case the partitioning uses the account number. Speed of access to the account information is improved by the use of an index.

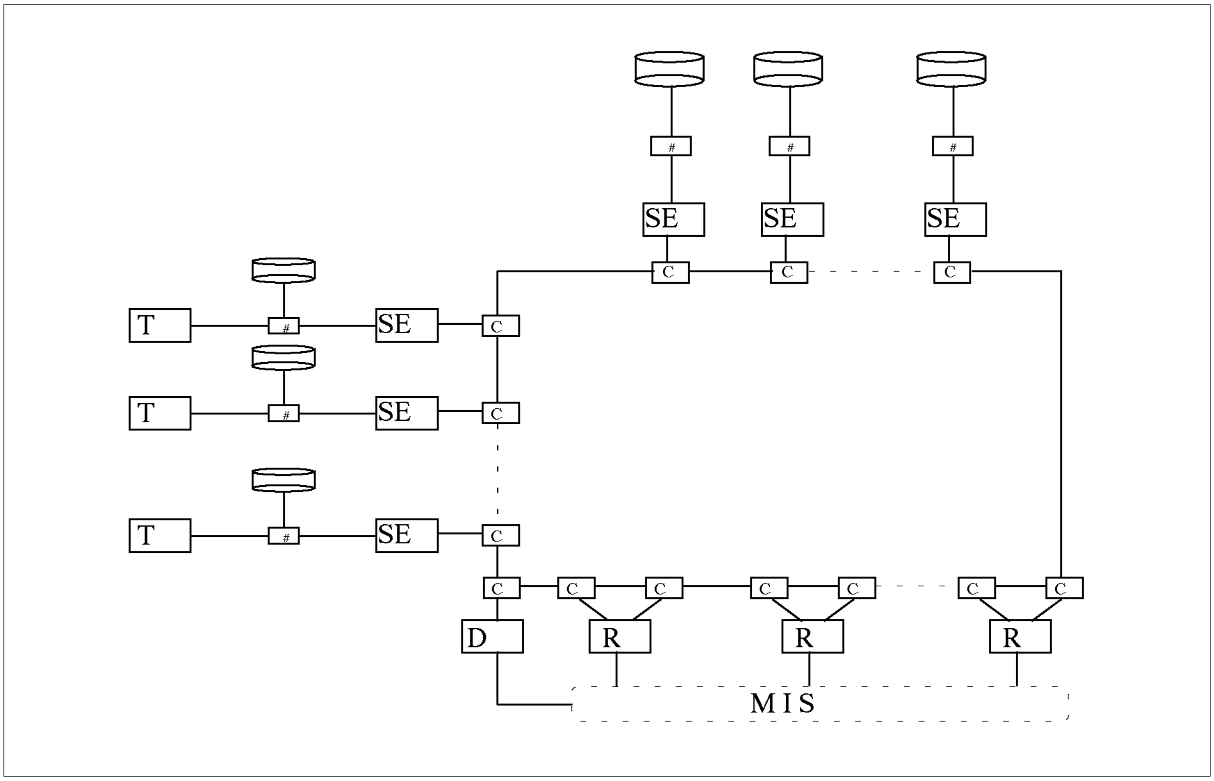

Figure 9.1 Basic IDIOMS architecture
Key:
T Transaction processor SE Storage engine D Data dictionary
R Relational processor C Communication engine # Disc controller

It is also presumed that the transaction processing time is small; that is, in general a transaction will access a single account, modify it in some simple manner and write the updated record back to disc. Conversely, it is presumed that a Management Information System (MIS) query will access many records in the database. and will thus take a long time to process. The Storage Engines connected to the Transaction processors are able to read data from the transaction data but not write data back. This means that an MIS query can be interrupted so that the T processor can access the disc, because this operation must be given priority. The IDIOMS machine design allows the transaction to access the data as if it were a traditional record structure and can thus be processed using a language such as C. The Storage Engine accesses the data as if it were SQL tables so that it can be processed in a relational manner. The machine design permits both operations concurrently on the same dataset. The overall design strategy is to ensure that the discs connected to the transaction processors (T) have sufficient spare access capacity to allow the amount of MIS activity required. The IDIOMS machine has demonstrated a transaction processing performance improvement of 45 times over the current mainframes used by TSB. The current system is incapable of providing MIS support. The demonstrator has shown that for the current mix of transactions there is sufficient disc access capacity available that the running of concurrent MIS queries results in no appreciable diminution of transaction processing performance [5].

The remaining Storage Engines are used to store data which is only accessed by the MIS system, for example summary and statistical tables. This data can be joined with the data held on the transaction discs in the relational processors R. MIS queries are input to the Data Dictionary (D) processor where they are parsed and processing resources are allocated as required. The data dictionary has sufficient information to know which parts of which tables are placed on which disc so that only those discs which hold data needed for the query actually contribute to the necessary processing. A sequence of relational operations can be constructed as a pipeline by sending the output of one Relational Engine (R) to the input of another using the communications ring of C processors. More details of relational processing techniques in such a machine can be found in

[6]. The network of C processors provides the scalability of the system because we can add extra nodes in the C processor structure as required. Thus we can add transaction nodes, MIS nodes and relational processing on an as needed basis. Compare this with a traditional mainframe solution where it is impossible to add the precise amount of extra capability required, rather the increment in performance quite often increases capability that did not need to be enlarged. In the following sections we shall discuss the changes that can be made to the IDIOMS design as a result of using T9000 and C104 technology.

9.3 A Processor Interconnection Strategy

Networks of up to 512 processors can easily be constructed using a simple three–level CLOS network (see figure 7.1). The network is replicated for each of the links of the T9000 if full interconnection is required. In the case of a database machine we may need to have more processors than this and we may also need to ensure that the original design permits easy on–site increase in size. Applications which can justify such processing needs usually cannot be taken out of service for long periods because they are critical to an organisation’s profitability. Figure 9.2 shows how a network of five–levels can be constructed which allows 1920 T9000s to be connected.

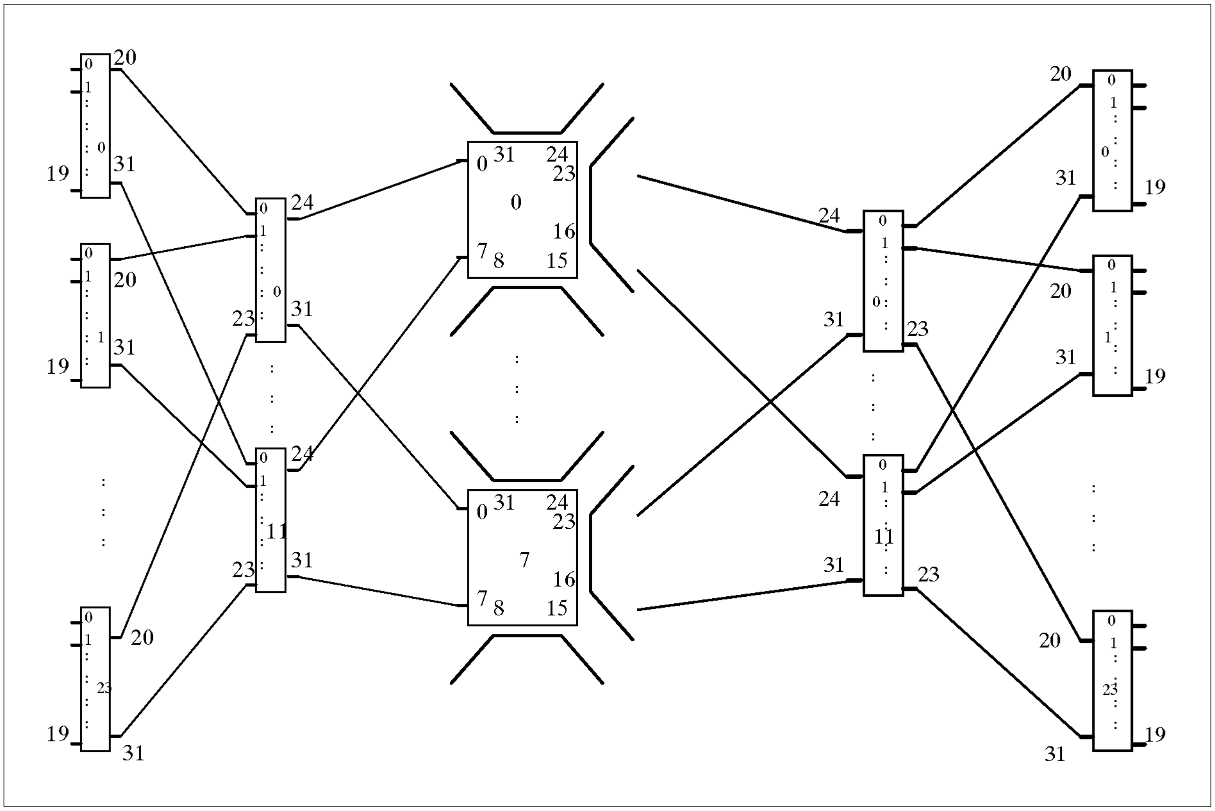

Figure 9.2 A five–level indirect network

The components in this network are all C104s. The terminal links are then connected to T9000s. The periphery of this network has sufficient capacity to hold 1920 T9000s each connected by a single link. If all four links are to be connected then the complete network has to be replicated four times. The element of the network to the right is duplicated and connected to the eight central C104s twice more, once for the lower connections and once to the upper connections. A total of 152 C104s are required to connect just one link of each transputer and thus 608 are required if all four links are to be interconnected. It should be noted that any communication between transputers on the same edge of the structure requires only three levels of communication rather than the five needed to cross from one edge to another. This structure gives sufficient capability for scalability once the database machine has been installed. The system needs initially to be set up with just one of the four quadrants and even that does not need to be fully populated. Thereafter the initial quadrant can be fully populated and subsequent quadrants filled as necessary. If only one quadrant is used then there is no need for the 8 central C104s.

9.4 Data Storage

Of crucial importance to any database machine is the provision of a high bandwidth, large volume, fault tolerant data storage sub–system. We chose to make the same design decision as was done in IDIOMS, namely that an operating system is not used to control the data storage because the file system is usually inappropriate for database operation. We therefore chose to store the data directly on the disc storage and use a Data Storage Description Language to specify the placement of the data[7]. This then permits greater and more flexible control of the database machine. Furthermore, the data dictionary process can utilize the information to make query processing more efficient.

In this design we propose to obtain fault tolerance by simply maintaining several copies of the data in a triple modular redundancy scheme. This is sometimes known as disc mirroring. We shall obtain high bandwidth by providing a large number of link connections to the disc subsystem. In some ways the design is similar to the many RAID (Redundant Array of Inexpensive Discs) products which are currently being marketed, except that we have chosen not to distribute the bits of a word over many discs. The design which is given presumes that a direct link interface to the disc unit is provided. Currently, of course, this is not the case, but the design gives compelling reasons why this should be done.

However before we can present the design a few basic facts about disc accessing are required. Disc manufacturers always quote a disc transfer speed which assumes that the read head is correctly located on the desired block before the transfer takes place. They also quote seek and latency figures which indicate the time taken to move the head to the correct track and to wait for the desired sector to rotate under the head. The figure they don’t quote is the effect of these times on overall performance. In experiments we have undertaken which are confirmed in another report[8] it was shown that an effective rate of about 0.5Mbyte/sec could be achieved from a SCSI–1 disc which had a rated performance of 3 Mbytes/sec. This was the figure for sequential access. The actual rate for random reads was of the order of 0.1 Mbytes/sec. Faster disc technology may improve this overall performance but the access rate is still going to be substantially less than the figure quoted by disc manufacturers. The way that disc manufacturers overcome this performance is by constructing disc strings, that is having a number of discs on the same bus, hence the SCSI bus system which permits upto seven discs on the bus. It has been found that the optimum number of discs to have on a SCSI–1 bus is four[9]. This figure matches the 0.5 Mbytes/sec and the rated performance of SCSI–1 of 2 Mbytes/sec, for sequential access. In order to achieve good performance in a disc array it is usually suggested that consecutive data blocks are placed on separate drives so that the seek and latency time can be overlapped. This works well if most of the accesses are sequential as happens for files in traditional operating system environments. However in a database system this is not the case and there is thus little likelihood of distributing disc blocks over drives having a beneficial effect. If such disc block striping were to be undertaken it would be best to do this over a string of drives connected to a single control processor. Figure 9.3 shows the structure of a simple disc sub–unit comprising 31 drives.

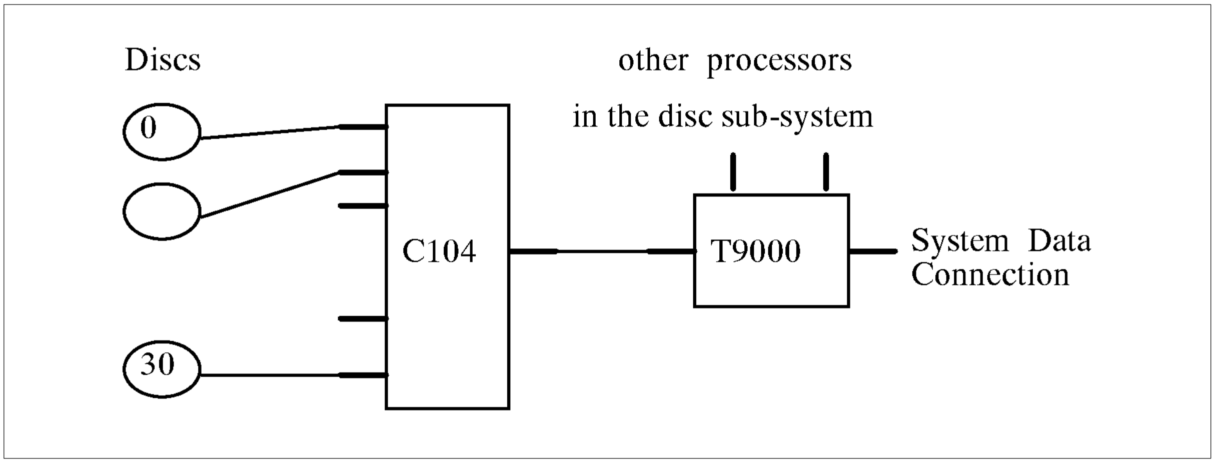

Figure 9.3 Disc sub–unit

The sub–unit chooses to have only one disc per connection to the C104. It is presumed that the disc drive contains an interface compatible with a T9000 link. In the short term this could be achieved by use of a standard disc with extra interface circuitry. The number of discs connected to a single T9000 link is justified because the bandwidth of a T9000 link is 17.48 Mbytes/sec bi–directionally. This capacity divided by the actual disc performance of 0.5Mbytes/sec result in up to 34 discs being reasonable. This sub–unit of itself has no fault tolerance and is not scalable. These aspects are achieved by making the sub–unit a component of a complete disc sub–system, as shown in figure 9.4.

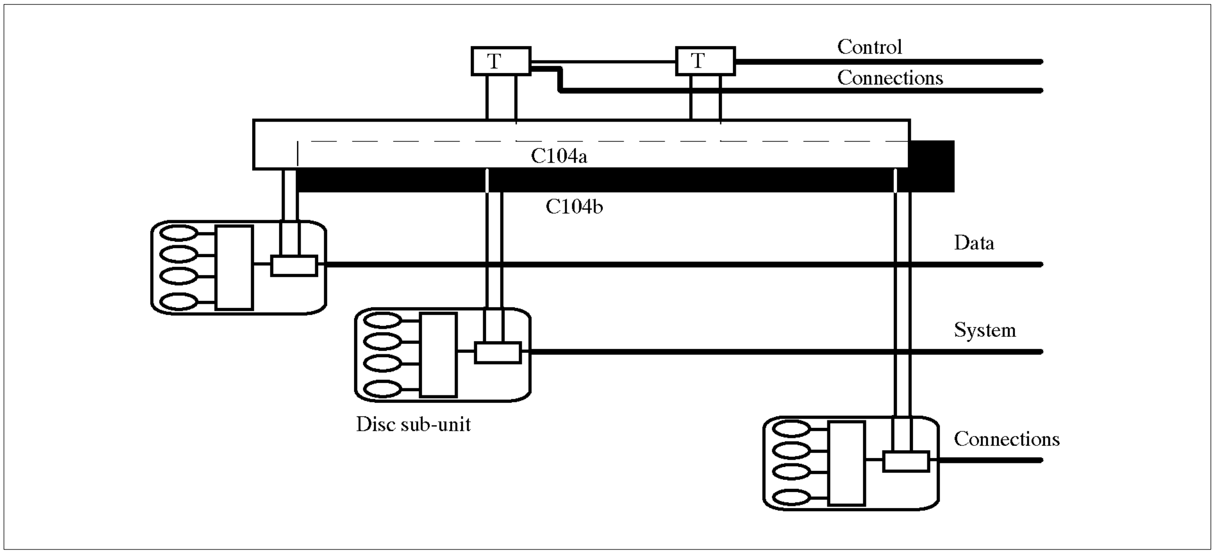

Figure 9.4 A complete disc sub–system

Each of the disc sub–units has one connection which connects it to the external environment. The other two connections are taken to a pair of C104s which provide connection between the sub–units. The two T9000's (T) which are also connected to the C104s are used to provide a fault tolerant repository of information about the data stored on the disc sub–system. The complete disc sub–system can comprise a maximum of 30 sub–units, though of course, it does not have to be fully populated initially. Assuming a fully populated system we can construct a disc sub–system which holds from 18Gbytes using 20 Mbyte capacity drives to 2325 Gbytes using 2.5 Gbyte capacity drives. In both cases, the bandwidth available is 524 Mbytes/second. As disc performance improves it will be necessary to reduce the number of discs connected to the C104 so that it matches the available link bandwidth. It should be noted that the capacity of the system will be reduced to one–third if a triple modular redundancy strategy is adopted.

Fault tolerance can be achieved by ensuring that every time data is written to the system two copies are sent via the sub–unit controlling transputer and the C104s to two other sub–units, where a copy of the data is kept. Thus we can be guaranteed that within one transfer time through a C104 data will have arrived at two other sub–units where the data can be replicated. At that point it may be necessary to wait to confirm the satisfactory writing of the data to all of the sub–units. A well understood two–phase commit protocol could be used to ensure system integrity. Read performance can be substantially improved because there are now three copies of the data. Even though a read request may be directed to a specified system connection link, there is no difference if the actual read is sent to a different sub–unit if one of the sub–units happens to be overloaded. The design could be criticized because there is only one link between the system connection and each disc. The effect of this weakness is however reduced because we have three copies of each data block, each on different disc units each having their own primary system connection. It is thus vital that we have a flexible interconnection strategy between the disc sub–system and the rest of the database machine.

9.5 A Disc Interconnection Strategy

Figure 9.5 shows the connection between the disc sub–system and the rest of the architecture when attached to an indirect network generated by 48 C104s, which permits 512 terminal connections.

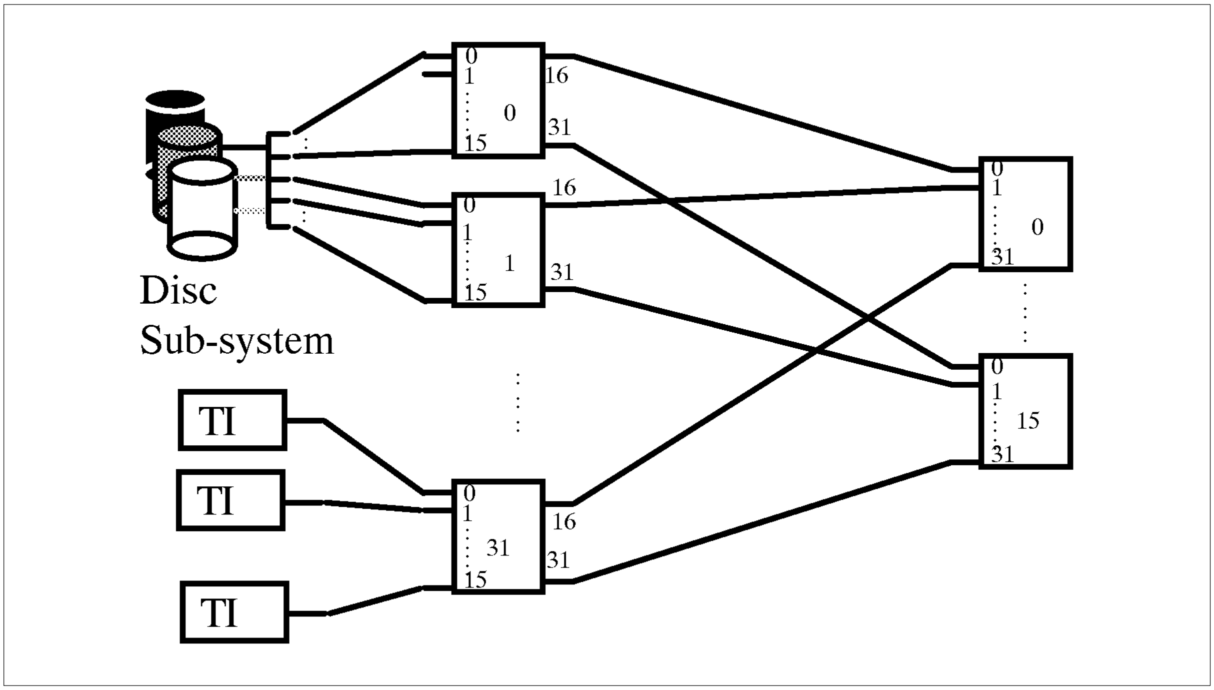

Figure 9.5 Disc sub–system interconnection

Each of the TI processors in figure 9.5 provide a generic Table Interface process to the disc sub–system. The disc sub–system is simply connected to the routing chips one link per terminal connection. This interconnection strategy permits the use of generic table handlers rather than the dedicated ones in the original IDIOMS design. Thus the table partitioning that was explicit in the IDIOMS design has become implicit in the T9000 based design. The table is allocated to the disc sub–system in such a way that the separate parts can be accessed in parallel by multiple TI processes. The TI process will usually have to manipulate the index that is used to access the part of the table that has been allocated to the particular TI process. A given query may not access the whole table and therefore only the required number of TI processes will have to be allocated to satisfy the table handling requirements of the query.

We now investigate how the remaining links on the TI process can be used given that the disc sub–system and the TI processes are on the same interconnection layer. First, we presume that the interconnection layers are replicated so that the transputers holding the TI process can be connected to other layers remembering that the disc sub–system is only connected to one layer. Thus we would end up with four layers of interconnection. We now have to allocate processes to these layers. It is not necessary in the connection system shown in figure 9.5 to consider locality of reference because all processors are equidistant from each other. In the interconnection architecture shown in figure 9.2 it would be necessary to consider which processes do communicate with each other so that those which communicate frequently are in a part of the network where there is a three level communication structure rather than one involving five levels. In the following sections we shall discuss the connections that have to be made between the processes that make up the database machine.

9.6 Relational Processing

Figure 9.6 shows the way in which the IDIOMS relational engines were constructed using three T8 transputers. This structure was required because it was necessary to provide some local buffering of data between the Storage Engine processors, which were sending data to the Relational Engine over the communication structure.

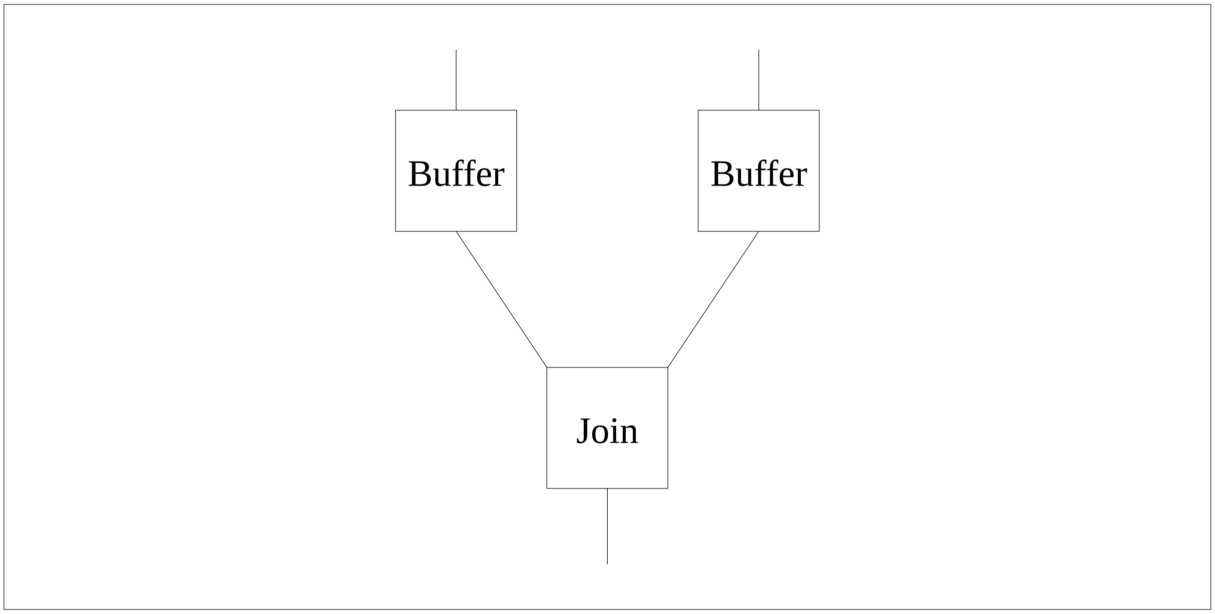

Figure 9.6 IDIOMS style relational engine

This design then imposed some software difficulties because the synchronization which normally occurs between occam processes is lost when that communication takes place between processes which are not on adjacent processors. This loss of synchronization can be overcome by having each message acknowledged by a special message which is sent from the buffer process to the storage engine which has sent the data. This extra communication results in a reduction in throughput because the sending process has to wait until it receives an acknowledgement before it can send the next block of data. The omission of the acknowledgement means that the buffer process has to be able to send messages to the storage engine, in sufficient time, so that data is not sent to the buffer process which cannot be stored in it.

This problem does not occur with the T9000/C104 solution because the hardware allows processes to communicate with each other directly. Thus the complete relational processor architecture can be implemented on a single transputer with the same process structure. However, the buffer process does not need to send wait messages to the sending process, it just does not input any more data when it becomes full, thus the sending process becomes blocked trying to output data. Provided the processes have been correctly constructed this causes no problem. The buffer processes are still required because it makes relational processing more efficient when a nested loops join has to be undertaken (every row of one table is compared with every row of a second table).

A general relational process can therefore be allocated to any one of the transputers in the architecture. In order to undertake the required processing the relational processor will need to be informed of the structure of the tables to be joined and the type of join processing to be undertaken. In addition, the relational processor will need to be told where the output from the relational processing is to be sent. This aspect of resource allocation and control of processing will be discussed in section 11.

9.7 Referential Integrity Processing

Figure 9.7 shows a typical situation that occurs in relational databases involving a many–to–many relationship between customers and their accounts. A many–to–many relationship cannot be directly represented so an intermediate linker table is introduced which implements two one–to–many relationships. The primary key of the Accounts table is the column A which contains the account number. The primary key of the Customer table is the column C which contains the customer identification number. The primary key of Account–Customer is a compound key comprising A and C, that is the combination of A and C is unique whereas individual values of A and C may be replicated. A fuller description can be found in [12]. A corollary of this structure is that in order to send letters to account holders it is necessary to join Accounts to Account–Customer on the common column A and then to join the result to Customer on the common column C.

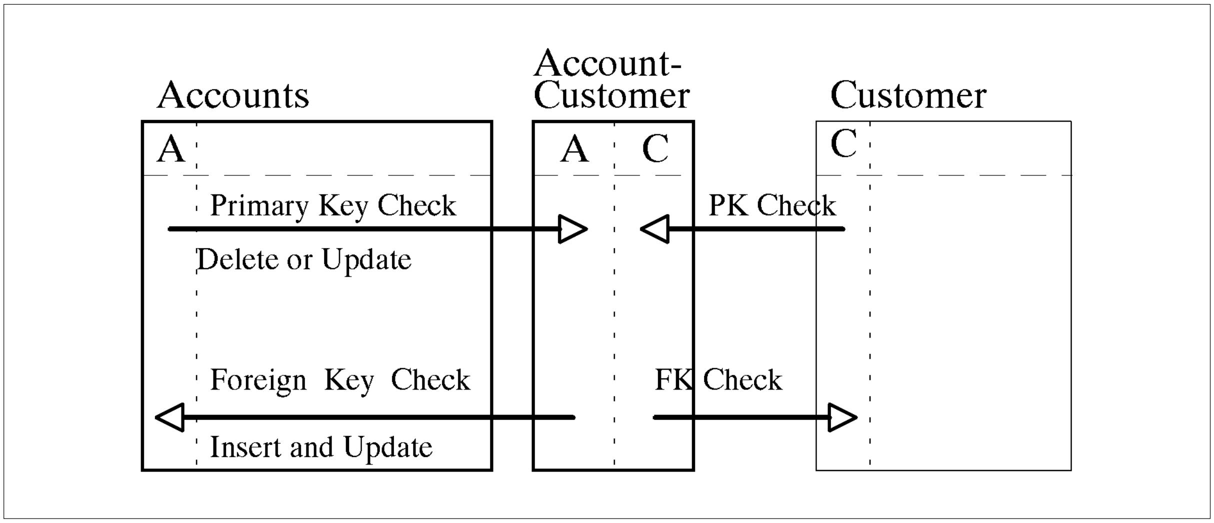

Figure 9.7 Foreign key, primary key relationships

Figure 9.7 shows the checks which have to be undertaken when undertaking insert, update and delete operations upon a database in which referential integrity processing has been specified. Thus, if it is desired to delete a row from either the Accounts or Customer tables, then it is first necessary to check that no row in the table Account–Customer has the same key value as that which is about to be deleted. That is the value of the column A or C respectively must have been deleted from Account_Customer before it is deleted from Accounts or Customer. Similarly, if a value of the primary key of Accounts A, is updated, then a check has to be made in Account–Customer to ensure that there are no rows which have the old value of A remaining.

Whenever a row is inserted into Account–Customer a check has to be made in both Accounts and Customer that a row with the same value for A and C already exist. This is known as a foreign key check. Similarly, if a row in the Account_Customer table is updated a foreign key check has to be carried out to ensure that the new values already exist in the referenced tables.

It is obvious from the foregoing description that much processing is involved in the checking of referential constraints especially in systems which involve much updating of data. It is for this reason that many existing database applications execute without referential processing enabled because the processing overhead is too great. Figure 9.8 shows how two co–operating processors can be used to implement a referential co– processing system.

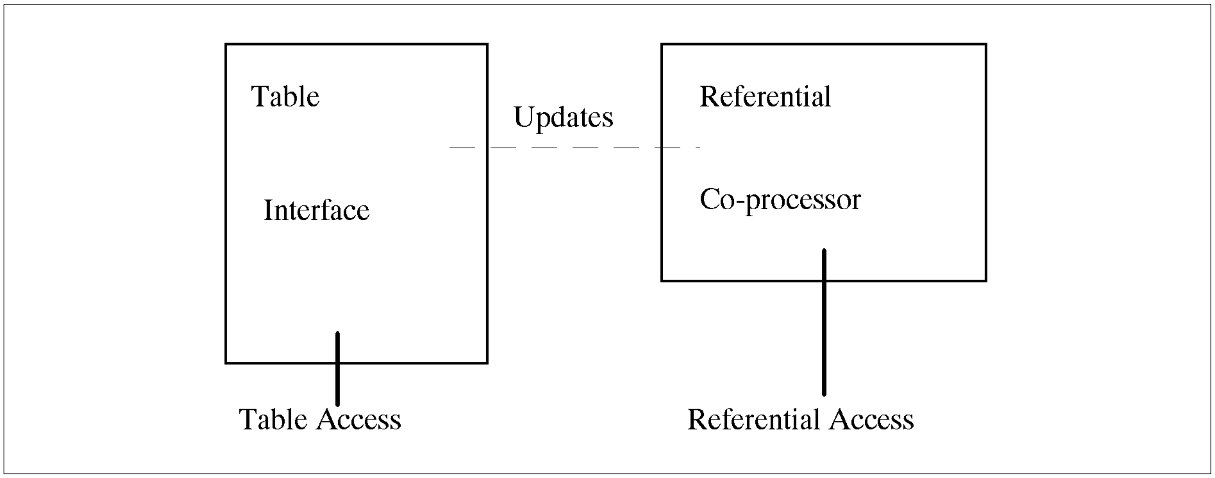

Figure 9.8 Referential co–processor architecture

The referential co–processor contains a copy, sometimes known as a concrete view, of the primary key column(s) of a table partition. This means that a particular referential co– processor is dedicated to a particular table partition and is not a general processor which can be allocated on an as needed basis like table interface processors. The referential co– processor can be accessed by any number of table interface processors because the access is read only as an existence check is being undertaken to check whether or not a value already exists in the referential co–processor. If a table interface process modifies the primary key of a table then those changes have to be communicated to the appropriate referential co–processors. This modification has to be done exclusively so that update anomalies cannot occur between table interface and referential co–processors. The referential co–processor is just a terminal transputer in the interconnect in just the same way as a table interface processor is connected. The only difference is that the referential co–processor undertakes the referential processing for a particular table partition. Thus, when a query is parsed that will invoke referential processing, access to the required referential co–processors will have to be granted.

The main advantage of this architecture is that the bulk of referential processing does not require access to the complete table, just to the columns which are referenced by other tables. It is thus sensible to provide this capability as a dedicated resource. The bulk of table accesses are, in fact, to read data from the table in response to queries, which need no referential processing. The disadvantage is that the data in the referential co–processor has to be up to date with all changes made to the database. This is closely linked with concurrency management which is discussed in the next section.

9.8 Concurrency Management

Figure 9.9 shows the architecture of the concurrency management system. Each Table Interface processor is a terminal processor in the interconnect structure as are the Transaction Manager processors (TM). The TM processors support one or more TM processes, though we shall assume this is just one for ease of explanation. There have to be as many TM processes as there are permitted concurrent transactions because we wish to ensure that the processing of one transaction is not disturbed by the processing of the other transactions which are running concurrently.

A transaction is a sequence of queries which a single user issues as an atomic piece of work. That is, either the whole transaction is successful and all modifications to the database are saved in the database, or the transaction fails and thus has no effect on the database whatsoever. A transaction may fail because a row from a table required by one transaction has already been allocated to a different concurrent transaction. It is a requirement of database management systems that they exhibit the principle of serializability. This principle ensures that the effect of a number of concurrent transactions is the same when executed concurrently as if they had been executed one after the other. In addition the effect of one transaction cannot be seen by other transactions until the transaction comes to an end and commits the changes to the database.

The design of this concurrency management system presumes that interference between transactions is low, which is reasonable for commercial style applications. For CAD/CAM applications this may not be justified and a different approach would be required because the nature of transactions is different, in particular, they tend to be much longer, which increases the likelihood of interference between transactions.

Each table is divided into a number of partitions to increase the parallel access to the table and to reduce the possibility of transactions interfering with each other. Each partition has its own, specific, Partition Manager process allocated to a dedicated processor which is connected to the interconnect in the same way as any other terminal processor. This process records which rows of the table partition have been allocated to which transaction. A Table Interface process determines whether or not it wishes to have access to a row. If it does require access to a row it sends a message to the Partition Manager associated with the table partition which the Table Interface process is accessing. At any one time many Table Interface processes may be accessing the same partition of a table. We have to ensure that these requests to access a row are received in a strict order. This can be simply achieved by using the Resource Channel mechanism provided by the T9000. This mechanism allows many processes to share a single channel which they can only access once their claim on that channel has been granted. This has a direct correspondence with the shared channel concept in occam3[10,11]. Figure 9.9 shows the individual shared channels with each Table Interface process having access to all the shared channels (indicated by the bold lines). There are as many shared Partition Control Channels as there are partitions in the database.

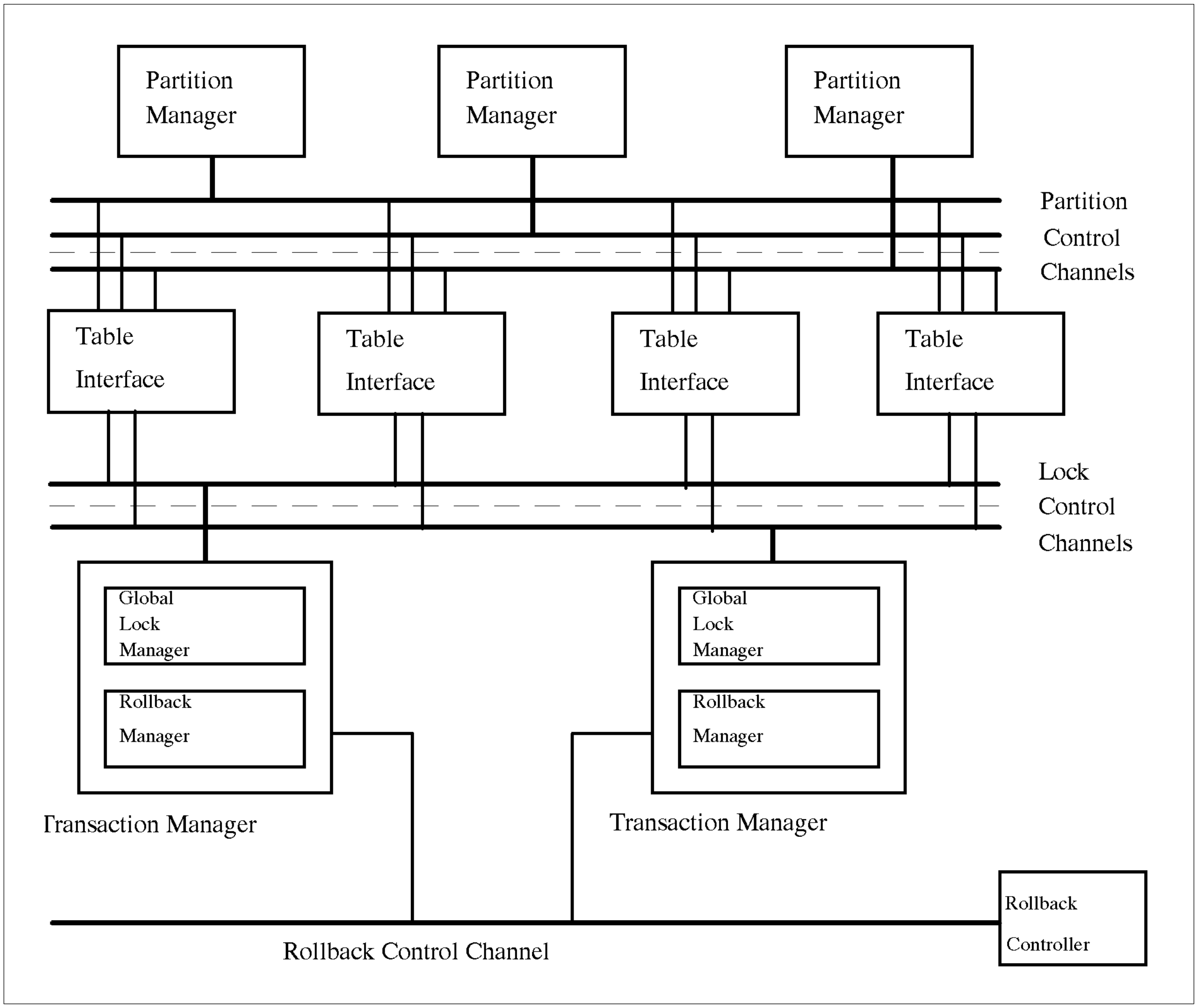

Figure 9.9 Concurrency management architecture

In addition, each Table Interface process has to indicate to one Transaction Manager process, with which it is associated, that it has gained access to a row of a table partition. If a Transaction Interface process attempts to access a row that has already been allocated to another transaction then the transaction becomes blocked and has to send a blocked message to its Transaction Manager. Yet again this mechanism has to ensure that access to the Transaction Manager is controlled and this can be simply achieved by the use of a resource channel. There are as many Lock Control Channels as there are Transaction Manager processes. Each Table Interface process can access all the Lock Control Channels.

The Partition Manager maintains a record of which rows of the associated table partition have been allocated to which transaction. The Transaction Manager maintains a record of those rows of table partitions that have been allocated to the particular transaction. In addition, the Transaction Manager needs to know with which other transactions it could interfere, so that it can determine if transaction deadlock has occurred. Two or more transactions are said to interfere with each other if they both access at least one table partition in common. In this case it possible that one transaction has already gained access to a row which the other transactions require. In this case the second transaction is made to wait until the first transaction commits its work. Transaction deadlock occurs when the transaction which is not blocked attempts to access a row which had been allocated previously to the other, now blocked, transaction. Neither transaction can make any progress because they are both waiting for each other to finish, which is impossible. This is just a simple deadlock; far more complex situations can happen in reality with many more transactions.

The traditional solution, adopted by most existing database management system implementations is to store all the lock information in a single data structure which allows the detection of such deadlock cycles. Necessarily, the access to this data structure, which is expensive to maintain becomes a bottleneck in the system. In the approach outlined above the amount of data that is saved for the normal situation, where no transaction blocking or deadlock occurs is very lightweight. It simply involves the communication of two sets of values from the Table Interface process, one set to the Partition Manager and the other to the Transaction Manager. In the normal case when the transaction completes successfully all the data structures (which are just simple lists of values containing no internal structure) will be emptied so that the memory space can be re–used for the next query.

If a transaction becomes blocked it has to determine whether or not a deadlock has occurred. This can be achieved by the Transaction Manager sending messages to other Transaction Managers with which it is known that the transaction interferes. If it is possible to construct a cycle amongst blocked transactions then it is known that deadlock has occurred and one of the transactions has to be rolled back. The cycle is created by following through each of the Transaction Manager processors looking at the row for which they are waiting. A cycle occurs when it is possible to return to the originating blocked transaction. A Transaction Manager can be informed which row it is waiting for and which transaction has accessed that row because that information is available in the Partition Manager. The decision as to which transaction to roll back is the function of the Rollback Control process. The system has been organized so that only one transaction can be rolled back at one time, hence the use of a resource channel between the Rollback Manager processes and the Rollback Controller processor, which is accessed by means of the shared channel Rollback Control.

9.9 Complex Data Types

It is becoming more important that database systems are able to support data types other than those traditionally supported by existing database management systems. Usually such systems are only capable of supporting integer, real, character and boolean data types. Some systems have supported date and time data types but in inconsistent ways. Some systems have also provided an unstructured data block into which a user can place a bit string of some length, which the user then manipulates as necessary.

The T9000 / C104 combination in conjunction with the occam3 provides a simple means of implementing complex data types through two mechanisms entitled remote call channels and library. A library allows a data type definition to be created with a functional interface to permit manipulation of structures passed to it using either ordinary channels or remote call channels. A library can be accessed by any number of concurrent user processes because it maintains no state information between calls to the library. A remote call permits the passing of parameters to a procedure using two implicit channels, one to send the parameters and the other to receive the results. It is similar in concept to the remote procedure call mechanism provided in some operating system implementations.

We can therefore construct a system in which one or more processors contain the code for a library which implements a particular complex data type. This library can then be accessed either using explicit channels or more likely by using remote call channels. The library is actually accessed using a resource channel which permits many user processes to access a single server process. The bottleneck of having a single processor to deal with a given library can be simply overcome by having many processors containing the same code and by using some form of resource sharing strategy. Resource channels can be passed as parameters so that a direct connection can be easily made by referring to a single process which allocates the resource. The complex data type processors are connected to the interconnect in the same way as any other terminal processor but once allocated are only able to process messages for a particular data type.

9.10 Recovery

In the IDIOMS environment recovery was undertaken at two different levels. The first dealt with recovery from storage media failure. This was achieved by simple disc mirroring. In the architecture described in this paper that aspect of recovery is dealt with by the disc sub–system using Triple Modular Redundancy and so can be ignored. The second type concerned recovery from transaction failure which occurs when there is some failure in the on–line transaction processing support infrastructure. Typically this occurs when there is a communication system failure. A transaction arrives at the computer system from a remote location, such as an Automatic Teller Machine, using a communications mechanism. If the communications media fails before the results of the transaction can be returned to the originating point, then the effect of the transaction has to be undone. There are a number of techniques which can be used to overcome this problem e.g. before images, shadow copies and transaction logs[12], which all require the saving of information on a stable storage media such as disc during the course of transaction processing. From the saved information it is possible to undo the effect of a particular transaction without having to re–instate the whole database. The architecture proposed in this paper can use these same techniques. Simply, a separate disc sub–system can be used to store transaction recovery information, automatically providing media failure recovery. A set of processors can be provided which can undertake the necessary processing to undo the effect of an incomplete transaction

9.11 Resource Allocation and Scalability

9.11.1 Resource Allocation

The IDIOMS architecture relied upon a single Data Dictionary / Parser processor which parsed incoming queries and allocated resources as necessary. As such it could become a bottleneck if the system was subject to a large number of small queries. The parsing of queries does not need to be restricted to a single processor. The parsing process entails the decomposition of a query into its component parts which can be allocated to separate processors for each query. A number of different processing strategies can then be identified which will depend upon the number of processors that are actually available when the query is resourced. The generation of these strategies can be undertaken without knowing what actual resources are available. In addition the strategies can be evaluated against each other to determine the most cost effective against some system defined cost function.

Once the strategies have been identified, the actual resources required can be communicated to a single processor which knows what resources are available. If one of the strategies can be accommodated then the resources can be allocated and the parser process can be sent information about the resources it can use so that it can send appropriate messages to the processors which will enable query processing to begin. When a query terminates a message can be sent from one of the processors to the single processor which holds resource availability information. If more than one strategy can be resourced, then the resource allocator processor can decide which strategy to use. The resource allocator processor could contain constraints which have to be met in order that a query can be started. It may be that at specific times of the day it would not be feasible to start a query which consumes most of the processing resource. For example, in banking systems it is known that there is a peak in transactions around lunch–time, hence it would be sensible to deny access to a large query which would use most of the processing resource just before midday.

Figure 9.10 shows a processor structure which will implement such a resource allocation strategy. We presume that queries arrive from the users into a User processor. The User processor then accesses the Resource Allocator process using the shared channel to determine which Parser process to use. If none are available the User process will be made to wait until one becomes available. The query is then sent to the indicated Parser process. It should be noted that all User processes are connected to all Parser processes. The Parser process then decomposes the query and determines the different strategies which are possible. The Parser process then accesses the Resource Allocator process using the shared channel Resource Request, which ensures that only one request for resources is dealt with at one time and thus it is not possible for the same resource to be allocated to more than one query. The Parser process will send information to the allocated resources, using channels not shown in the diagram, indicating the processing to be undertaken. Generally results will be returned to the User process from the Relational processors (R), hence it is necessary to connect all the R processors to all the User processors. When the query is complete the User process will send a message using the shared channel which accesses the Resource Allocator process to indicate that the resources used by the query are no longer required and can be allocated to another query.

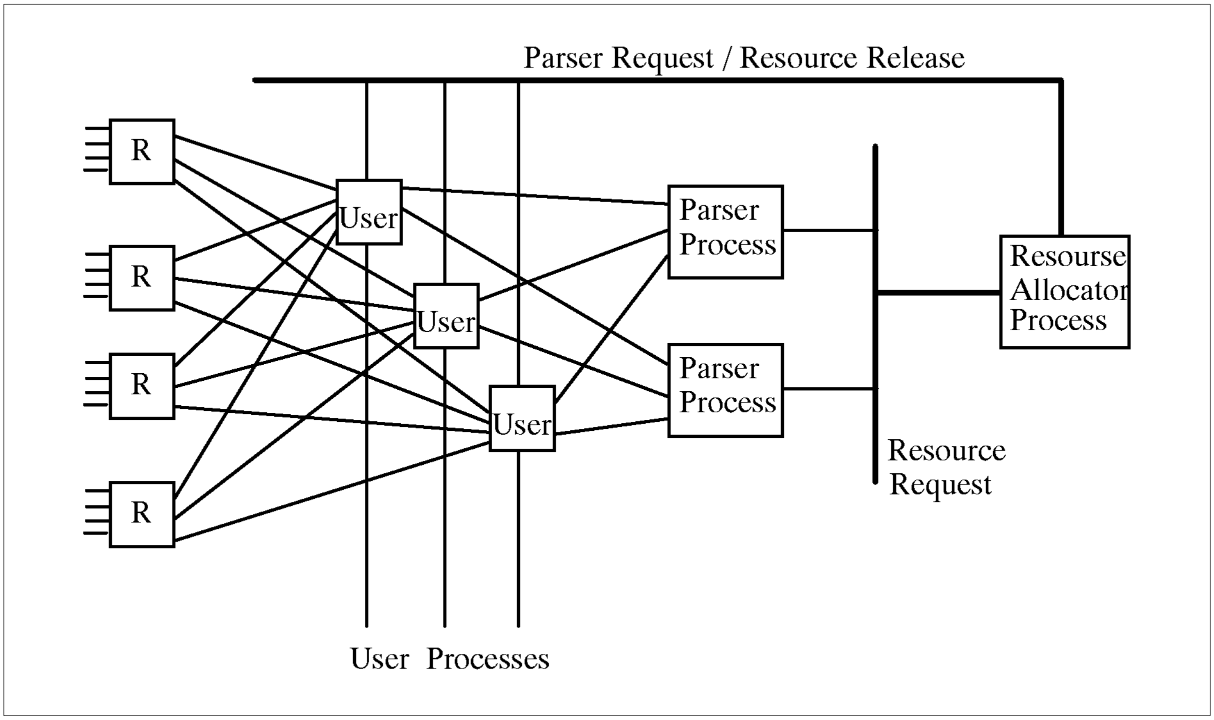

Figure 9.10 Resource allocation processor structure

9.11.2 Scalability

The system described in this paper is scalable in the two ways identified previously. First, the installed size of a system can be matched with the initial system requirements. In coming to this initial size the system designer must be aware of the likely increases in storage and performance that will ensue. For example, it is not uncommon for system to double in storage requirements over the first two years with a consequent increase in processing requirements. Thus it is vital that the system interconnect is designed so that the perceived increases can be accommodated. It is thus not sensible to build an interconnect that is limited to 512 terminal connection points if it can be anticipated that more will be needed in the future.

Secondly, the system can be scaled after installation by simply adding further resources. These resources can be added wherever they are required within the functional components in the machine because there is a uniform interconnect mechanism with a known cost. The only constraint would be in the five–level indirect structure, shown in figure 9.2, where it may be preferable to add some facilities within a three–level interconnect regime to ensure the required performance. In adding extra resources the only part which has to be changed is the resource allocator process discussed previously. Each component in the architecture that has been described is essentially a generic component, even if in use it is made specific to a particular task, such as the referential co–processors. This means that no new software has to be constructed. Thus the implementation of the system as a highly parallel system has afforded an easy mechanism for scalability.

A key factor in the operation of the database machine will be the collection of statistics so that optimal data storage can be achieved. A vital component of the collection of statistics is the monitoring of the changes in queries with time as the use of the database develops. We have already started work on such an automated system[13].

9.12 Conclusions

This paper has presented the outline for the design of a highly parallel database machine which is solely dedicated to that single task. The use of a general purpose processor has been avoided thereby ensuring that the design has had to make few compromises concerning the implementation. The advantage bestowed by the T9000/C104 combination is that we can design each individual software component as a stand–alone entity which makes the system inherently scalable. A further advantage of the use of these hardware components is that the resulting interconnect is uniform in the latency that it imposes upon the system thus the system designer does not have to take any special precautions to place closely coupled processes on adjoining processors. The paper has also shown how it is possible to build a highly parallel disc sub–system. It is a subject for further research to best determine how data should be allocated in such a system in order to maximize parallel access to the data stored in the disc sub–system. Undoubtedly, the use of a Data Storage Description Language[14], such as that developed for the IDIOMS project will be required.

Acknowledgements

The ideas expressed in this chapter are those of the author but necessarily they result from discussions with a large number of people and are also due to interaction with real users of large commercial database systems. The author is indebted to the discussions held with Bill Edisbury and Keith Bagnall of TSB Bank plc and Bob Catt, Alan Sparkes and John Guast of Data Sciences Ltd. The co–workers within the University of Sheffield include; Siobhan North, Dave Walter , Romola Guiton, Roger England, Paul Thompson, Sammy Waithe, Mike Unwalla, Niall McCarroll, Paul Murray and Richard Oates. The work discussed in this paper has been supported in part with funds from the UK Science and Engineering Research Council (through CASE Awards) and the UK Department of Trade and Industry.

References

1. RJ Oates and JM Kerridge, *Adding Fault Tolerance to a Transputer–based Parallel Database Machine*, in Transputing '91, PH Welch et al (eds), IOS Press, Amsterdam 1991.

2. RJ Oates and JM Kerridge, *Improving the Fault Tolerance of the Recovery Ring*, in Transputer Applications '91, T Duranni et al (eds), IOS Press, Amsterdam, 1991.

3. JM Kerridge, *The Design of the IDIOMS Parallel Database Machine*, in Aspects of Databases, MS Jackson and AE Robinson (eds), Butterworth–Heinemann, 1991.

4. R England et al, *The Performance of the IDIOMS Parallel Database Machine*, in Parallel Computing and Transputer Applications, M Valero et al (eds), IOS Press, Amsterdam, 1992.

5. JM Kerridge, *IDIOMS: A Multi–transputer Database Machine*, in Emerging Trends in Database and Knowledge–base Machines, M Abdelguerfi and SH Lavington (eds), to be published by IEEE Computer Science Press, 1993

6. JM Kerridge, *Transputer Topologies for Data Management*, in Commercial Parallel Processing and Data Management, P Valduriez (ed), Chapman and Hall, 1992.

7. JM Kerridge, SD North, M Unwalla and R Guiton, *Table Placement in a Large Massively Parallel Database Machine*, submitted for publication.

8. AE Eberle, *A Gem of a Disc Drive*, Digital Review, Cahners–Ziff Publishing, January 14 1991,

9. V Avaghade, A Degwekar and D Rande, *BFS – A High Performance Back–end File System*, in Advanced Computing, VP Bhatkar et al (eds), Tata McGraw Hill, 1991.

10. G Barrett, occam3 *Reference Manual Draft (31/3/92)*, Inmos Ltd, 1992

11. JM Kerridge, *Using* occam3 *to Build Large Parallel Systems : Part1;* occam3 *Features*, submitted for publication

12. R Elmasri and SB Navathe, *Fundamentals of Database Systems*, Addison–Wesley, 1989.

13. M Unwalla and JM Kerridge, *Control of a Large Massively Parallel Database Machine Using SQL Catalogue Extensions and a DSDL in Preference to an Operating System*, in Advanced Database Systems, PMD Gray and RJ Lucas (eds), Springer–Verlag, LNCS 618, 1992.

14. JM Kerridge et al, *A Data Storage Description Language for Database Language SQL*, Sheffield University, Department of Computer Science, Internal Report, CS–91–05, 1991.