TR-83-007

# Efficient Computing of Relational Join Operations by Means of Specialized Hardware

BY

Yang-Chang Hong*
Institute of Information Science
Academia Sinica
Taipei, Taiwan, ROC

February 14, 1983

*This work was supported by the National Science Council of the Republic of China, Taipei. The author is currently visiting the Department of Electrical Engineering and Computer Science, University of Santa Clara, Santa Clara, CA 95053.

Telephone: (408) 984-9017

## ABSTRACT

The paper describes a hardware architecture which can provide powerful join capabilities to associative processing (AP) systems. The main feature of the hardware is a bit- and word-addressable store, RAM, which can rapidly remember or recall data. The data might be the values/tuples selected from one relation in which case the RAM helps on performing the joining of these values/tuples with the tuples in the second relation. For the general case of the join, the RAM can help on dividing the tuples of the relations being joined into buckets according to different value-intervals of their join columns. An array of servers is introduced for producing the concatenated tuples of this join. This hardware design emphasizes on much parallelism in the cross referencing involved in the join operation, giving considerable performance improvement over existing AP systems. The paper finally gives the analysis result of the hardware performance under different applications.

Key Words and Phrases: implicit join, explicit join, associative hardware, associstive processing systems, join columns, referential integrity.

## 1. INTRODUCTION

The relational model [5] has, more than other models during the past decade, attracted and held great interest of the database researchers and data processing community. But efficiently implementing the model on main frame computer (MFC) is rather a challenging problem up to date.

Several hardware approaches to implementing the relational database have been reported in the literature [1-19]. Previously, the design of associative hardware for the joining of relations has been to concentrate on a form so called the "implicit" joins [1, 3, 11, 12, 16]. This join does not create a derived relation; instead the values selected from one relation are transferred to select the tuples in the second (or the same) relation that have the same values in their join columns (i.e., the columns on which their joining is based). The early designers of associative processing (AP) systems did not provide powerful facilities for the explicit (as opposed to the implicit) join operation. The explicit join algorithms they provided are mainly carried out by the MFC to which the AP system is a backend. The LEECH [9] and CAFS [1] machines use a filter for selecting tuples needed for the join. The selected tuples are sent to the MFC to form the concatenated tuples of the join. The only difference of these two machines is the design of the filter. RARES [11] provide a hardware-support algorithm for dividing the tuples of the relations being joined into buckets according to different value-intervals of their join columns. The tuples within each bucket are sorted in the main memory and the sorted buckets are used for computing the join by MFC. These algorithms will not be very effective, especially when the large number of tuples being joined is involved.

Since AP systems are based on the parallel processing of the segmented sequential search, while the explicit join operation requires involving a great amount of cross checking which breaks this parallelism, they are not alone sufficient to make a high-performance database machine. New hardware which can perform a large amount of cross checking in parallel must be sought to cope with a join-dominating database application.

There have been at least three hardware architectures proposed for the explicit join operation [10, 12, 18]. They provide the implementation of the general form of explicit joins. Our study, however, shows that the explicit joins which perform the joining of the tuples via the keys - candidate and foreign - can be implemented in a more efficient way than the general form. Also, from the viewpoint of data semantics, one can say that a majority of explicit joins are performed via the keys. It is, therefore, advantageous to refine this type of operation further for implementation.

This paper describes a hardware architecture which can provide powerful join capabilities to AP systems. The architecture implements three types of join: the implicit join, type I and II explicit joins. The type I explicit join is the join defined above and type II is the explicit join in which the joining of the tuples is not via the keys. The main feature of the hardware is a bit- and word-addressable store, RAM, which can rapidly remember or recall data for the implicit join and the type I explicit join. In the type II explicit join, the RAM can help on dividing the tuples of the relations being joined into buckets according to different value-intervals of their join columns. An array of servers is introduced to produce the concatenated tuples of this join. This hardware design emphasizes on much parallelism in the cross referencing, giving considerable performance improvement over existing AP systems. A hardware simulator is developed on PDP-11/70 for determining the major design parameters, such as the number of servers, the length of queues associated with servers, etc. It is also used to (given a fixed number of servers) determine how good the hardware performance will be under different applications.

The body of the paper is divided into three parts. In the first part the hardware architecture is described. The second part is concerned with the algorithms for three types of joins. The third part is concerned with the analysis results of the proposed hardware, which is followed by a summary and conclusion.

## 2. HARDWARE

The hardware (see Figure 1) described here provides join capabilities to an AP system. It accepts a sequence of column values/tuples from the AP system where data are searched in parallel by the search logic. The command and control processor (CCP) receives data requests from the MFC, translates them into commands for both the AP system and the hardware, distributes commands for execution, receives the data transferred out of the AP system, and outputs the data to the MFC. In the discussion which follows, data stored in an AP system are in coded form and the encoding and decoding process are done by CCP.

The hardware consists of five major components - IP, MB, RAM, S, and CP - as shown in Figure 1 where

(1) IP is an input processor which accepts column values/tuples from the AP system and stores them in queue Q. The queue Q acts as a buffer between the AP system and the hardware. Associated with Q are two registers, T and H, and one flag $F_Q$. The T- and H-registers are used to hold the locations of the last and first entries of the queue. The setting of the flag $F_Q$ indicates the queue Q is full. This will also notify the AP system to stop outputting data to the hardware. The IP will start its processing once the flag $F_Q$ is clear.

(2) MB is a memory bank for holding the tuples of relations being joined. It consists of p memory modules M(i), $0 \leq i \leq P-1$, and each module has q words, where p and q are design parameters.

(3) The RAM is composed of one or several single-bit, directly addressed stores rA, rB,..., (note: Two stores are sufficient for joins. Others are provided for projections) and an array r of words. It is used to hold intermediate results. The single bit array store is addressed by encoded values and can hold encoded values, counters, and pointers (to be

detailed later). If a pointer word is concerned, it can point to a specific word of a specific memory module. Such a word can be regarded as consisting of two parts; one storing a value pointing to a particular memory module and the other storing a value pointing to a particular word within the module. Thus, the array r of words can be seen as consisting of two arrays, r' and r", of words and the sizes of the word in r' and r" are $\lceil \log_2^P \rceil$ and $\lceil \log_2^q \rceil$, respectively, where $\lceil x \rceil$ is the least integer greater than or equal to x.

(4) S is a set of queue servers $S_i$ each associated with a queue $Q_i$. The $Q_i$'s are served to hold incoming tuples of the second relation being joined. Like queue Q, each $Q_i$ has two registers, $T_i$ and $H_i$, and one flag $F_i$. Each queue server $S_i$ is designed to read data from its queue $Q_i$ and the memory module M(i). Thus, there are as many $S_i$'s as M(i)'s. A buffer is provided for each $S_i$ for holding the results produced, which are either output to the MFC or stored back to the AP system for further processing. The data transfer is accomplished by an output mechanism.

(5) CP is a central processor which fetches column values/tuples from Q and uses them as indices to address the bit array store for setting to 1 or 0, or testing for being 1 or 0, or to locate desired words in r for various purposes (to be detailed later). The CP also serves to allocate storage space in MB for storing tuples of the relation being joined. The registers T, D, and BR(i), $0 \leq i \leq P-1$, are provided for storage space allocation, and are best explained when used.

## 3. IMPLEMENTATION OF RELATIONAL JOINS

This section shows how the hardware performs relational joins. We will first consider the implementation of implicit joins and type I explicit joins and then consider type II explicit joins.

### 3.1. Queries Involving the Implicit Joins

Implementing an implicit join by the single bit array stores is best explained by means of an example.

Example 1. Print all the green items sold by the D1 department.

To answer this query, a simplified database with tables SALES and TYPE is assumed in Figure 2. This query can be implemented by various ways. One way is to apply the selection process to the table SALES to select the items sold by the D1 department. The selected items are then transferred to the table TYPE as a disjunctive condition to retrieve all the green items. The procedure can be implemented by using the store rA, which is outlined below:

(1) Clear the single bit array store rA.

(2) Scan the table SALES by the AP system and output the items sold by the D1 department to the input processor IP. The items fed to IP are then queued in Q. They are then fetched by CP and used as indices to address the store rA and recorded in rA.

(3) Scan the table TYPE by the AP system and output and store all the green items in Q. Any item in Q is output to the MFC if it has been recorded in rA, i.e., it is an item sold by the D1 department.

Here we assume that the reader is familiar with the data search performed by the AP system. What is not made clear is the function of the single bit array store rA; how the CP records the items in rA and how it determines which green items are to be output to the host.

Recall that data are encodedly stored in the AP system. Each bit position in the array can be made corresponding to an encoded value. With this technique, the addressed bit can be set to 1 or 0, or tested for being 1 or 0. We give a real example to illustrate this technique.

Assume that BOLT is encoded as 0, i.e., <BOLT> = 0, and <CAM> = 1, <COG> = 2, <GEAR> = 3, <NUT> = 4, and <SCREW> = 5. At the end of step (2), the bit pattern of rA will be (0,1,0,1,0,..., 0). This pattern would record the list of items CAM and GEAR. In step (3), items <BOLT> and <GEAR> are selected and stored in Q for examination. Since rA(<BOLT>) = rA(0) = 0, <BOLT> is discarded. Similarly, rA(<GEAR>) = rA(3) = 1, <GEAR> is output to the MFC. Before <GEAR> is output, it is decoded by the encoding and decoding unit (EDU) in CCP. Of course, values D1 and GREEN in the query have to be encoded by the EDU before the query is executed. (We neglect the detailed encoding and decoding processes here.)

The discussion above assumes that all the encoded ITEM values are within the address space of rA. If not, they are divided into buckets; the values in the first bucket lie between 0 and $2^t - 1$; the values in the second bucket lie between $2^t$ and $2^{t+1} - 1$; and so forth, where t is the number of bits required in the address space. Each bucket is then evaluated by repeatedly applying the same procedure being described.

The idea of using the single bit array store to remember or recall data is the same as those used in CASSM [16, 17] and CAFS [1]. CASSM uses a single bit array store per cell (consisting of a memory element and a processing logic) and one logical single bit array store, consisting of the concatenation of single bit array stores of cells, addressable by each processing logic. To address a bit in the logical array store requires passing the bit address (i.e., the encoded value) from one logic to another. Moreover, only one cell is allowed to address the logical array store at one time. If two cells want to address the store simutaneously, one of the two cells must wait for the subsequent revolution. This means additional memory revolutions (or scans) are required in addressing the bit array store. Our approach, like CAFS, uses a central processor CP to set or test a single bit array store, thereby eliminating memory addressing contention. Because of the use of an AP system, which acts as a filter, less data than CAFS are fed to the CP for setting or testing the bit array store.

If the values selected from the second relation are transferred to select tuples in the third relation, the second single bit array store is needed. In general, two stores are sufficient and can be used alternatively for a query involving a chain of implicit joins.

## 3.2. Type I Joins

By extending the concept of single bit array stores to an RAM, type I explicit joins can be implemented as effectively as the implicit join. Two examples below are used to illustrate this.

Example 2. Find the names of the employees who make more than their department managers. The query is directed at the table

EMPLOYEE(NAME,SALARY,DEPT,MGR)

where the managers are also employees - i.e., the values in the MGR column also appear in the NAME column.

One way to answer this query is first to scan the MGR column and output unique managers. Next select EMPLOYEE tuples where NAME = 'one of the selected managers' and then join the tuples being selected with EMPLOYEE tuples that have the same as those names in their MGR columns. Finally, scan the joined relation and output the employee names whose salaries are greater than their managers. This method performs an implicit join followed by an explicit join and a selection operation. It is obvious that single bit array stores alone are not sufficient to remember the manager names and their salaries for being used to select those employee names who make more than their managers. Our approach which uses the single bit array store rA and the array r of words for storing the manager names and their salaries is outlined below:

(1) Clear rA.

(2) Scan the EMPLOYEE table and output and store the entries in the MGR column in Q. The entries in Q are then fetched and used to set the rA bits.

(3) Scan the EMPLOYEE table again and output and store the employee names and their salaries in Q. Fetch each pair (<name>, <salary>) in Q and test if rA(<name>) is 1. If rA(<name>) = 1, then store the salary in the corresponding r-word, i.e., r(<name>) ← <salary>. Otherwise, discard the pair.

(4) Scan the EMPLOYEE table again and output and store the employee names, salaries, and managers in Q. Fetch each triple (<name>, <salary>, <manager>) in Q and test if r(<manager>) < <salary>. If yes, output the <name>. Otherwise, discard the triple being held.

We notice that the encoded values in the SALARY column should have the same order as they originally have. This procedure combines one Type I explicit join (this is the case where the two relations being joined are not distinct) and one selection operation to a single process where the manager names and their salaries are recorded in RAM and each incoming EMPLOYEE tuple is virtually concatenated to a proper entry in RAM so that the qualified employee names can be determined immediately. Our observation concludes that this technique can be applied to the joins in which their join columns satisfy the referencial integrity [5]. The Example 2 illustrates the case where the two relations being joined are not distinct. The Example 3 below is the case where they are distinct.

Example 3. Join the tuples of the SALES table with those TYPE tuples having items whose price is greater than 4P and output the DEPT, ITEM, and COLOR columns. This is a typical explicit join of two relations. The column ITEM in table SALES is sometimes called the foreign key. This join can be realized by the following procedure:

(1) Clear rA.

(2) Scan the TYPE table and output and store the items and their color in Q if their price is greater than 4P. Fetch each pair (<item>, <color>) in Q and record it in RAM. That is, rA(<item>) $\leftarrow$ 1 and r(<item>) $\leftarrow$ <color>.

(3) Scan the SALES table and output the SALES tuples to Q. Fetch each tuple (<department>, <item>) in Q and test if rA(<item>) = 1. If yes, concatenate r(<item>) to the tuple being held and output to the MFC. Otherwise, discard it. If further processing is needed each new tuple is stored back to the AP system.

If entire relations SALES and TYPE are joined, there are at least two ways to implement this join. The first way is to divide it into two steps: the first one is to join SALES and $\text{TYPE}^{(1)}$(ITEM, COLOR) over ITEM, denoting the resulting table as R1, and the second one is to join R1 and $\text{TYPE}^{(2)}$(ITEM, PRICE) over ITEM. Each step follows the same procedure as described above. The second way is to treat each word in the array r as a pointer word pointing to the starting address of a block of words in MB in which a TYPE tuple is stored, except for its identifier. This approach would modify the step (2) of the above procedure as

(2') Output and store TYPE tuples in Q. Read each tuple and store it, except for its identifier, t-id, say, in a block of words in MB. The starting address of the block is then stored in r(<t-id>) and the tuple identifier is recorded in rA(<t-id>). The addresses recorded in r are provided for step (3) for locating TYPE tuples.

The latter approach is generally better than the former one. It is noticed that in type I explicit join, the relation whose join column is a candidate key must be scanned first and the output data items are recorded in RAM.

## 3.3. Type II Explicit Joins

### 3.3.1. General Description

Type II explicit join may make the implementation rather costly in time and storage. The cost in storage is reduced by dividing "large" relations being joined into buckets according to their join column values, in such a way that the first bucket of the first relation is to join with the first bucket of the second relation; the second bucket of the first relation is to join with the second bucket of the second relation; and so forth. A relation is "large" if it satisfies one of the following conditions: (1) The range of the encoded values of its join column exceeds the address space of RAM and (2) it cannot be entirely stored in the memory bank MB. The processing time is decreased by increasing the parallelism of the cross referencing. This parallelism is achieved by further dividing the tuples of buckets being joined into sub-buckets. The first sub-bucket of the ith bucket is then joined with the first sub-bucket of the jth bucket; the second sub-bucket of the ith bucket is joined with the second sub-bucket of the jth bucket; and so forth, where ith and jth buckets have the same value-interval. The join of the pairs of sub-buckets is done in parallel by the array of servers in our approach. Although the tuples in the sub-buckets are not actually sorted in the order of join column values, however, it will be seen that, in logical effect, they are joined from the "sorted" sub-buckets in our approach.

In our design, the division of the large relations being joined into buckets is relegated to the AP system, similar to RARES [9], so that fewer tuples are output to the hardware. The pairs of buckets are then sent to the hardware, one pair at a time, for computing the join. The hardware uses two single bit array stores rA and rB for first filtering out the irrelevant tuples of the join since the join column values in one bucket may not appear in another. It is worthwhile to do so, especially when a large number of irrelevant tuples are involved. The rA, rB, and r, except for helping with Type I explicit joins on remembering or recalling data as described previously, can also help with type II explicit joins on dividing tuples of each bucket into sub-buckets. For one pair of buckets being joined, the sub-buckets of one bucket are first stored in the memory modules M(i) of MB, one per module. Each incoming tuple of the second bucket is then stored in the corresponding queue $Q_i$; that is, the first queue $Q_1$ accepts only those incoming tuples whose join columns have the same value-interval as those stored in M(1); the $Q_2$ accepts only those incoming tuples whose join columns have the same value-interval as those stored in M(2); and so forth. This arrangement permits each queue server $S_i$ $(0 \leq i \leq p-1)$ to produce the concatenated tuples of the join from its queue $Q_i$ and the M(i) in parallel, without any memory addressing contention. What is not made clear here is how each $S_i$ can know which tuples in M(i) are concatenated to the tuple being fetched from $Q_i$. This can be seen from the following algorithm.

## 3.3.2. Algorithm for Explicit Equi-Joins of Two Buckets

Let us denote the two buckets being joined as $R_A$ and $R_B$ of relations A and B with $A = (X_1, X_2, \ldots, X_u)$ and $B = (Y_1, Y_2, \ldots, Y_v)$, respectively, where $X_i (1 \leq i \leq u)$ and $Y_j (1 \leq j \leq v)$ are column names. Assume that columns $X_a$ and $Y_b$ are of the same underlying domain. The following algorithm is to compute the join of buckets $R_A$ and $R_B$ over $(X_a = Y_b)$. The resulting table consists of the set of tuples t, where t is the concatenation of a tuple t' belong to $R_A$ and a tuple t" belong to $R_B$ and $x_a = y_b$ ($x_a$ being the $X_a$-component of $R_A$ and $y_b$ being the $y_b$-component of $R_B$). The algorithm is outlined below:

(1) Initialization: Clear rA, rB, and r .

(2) Output $X_a$-components of $R_A$, set the rA to 1, and increment the corresponding counter words of r : Clear Registers, T and H, and the flag $F_Q$ of Q. Scan the relation A and output the sequence of $X_a$-components $x_a$'s of $R_A$ (in encoded form) to IP. The IP accepts each component $x_a$ and deposits it into Q. The $x_a$'s in Q are then fetched, one at a time, and used as indices to address the bits in rA and the corresponding counter words in r. The addressed rA bits are set to 1 and the corresponding counter (or r) words are incremented by 1. If $x_{ai}$ is fetched, for example, then $rA(x_{ai}) \leftarrow 1$, and $r(x_{ai}) \leftarrow r(x_{ai}) + 1$. At the end of step (2), the word $r(x_a)$ contains a value indicating the number of $R_A$ tuples with $X_a = x_a$ . The discussions which follow use [r(x)] to denote the contents or value of the r word addressed by x .

(3) Output $Y_b$-components in $R_B$, set the rB to 1, and allocate memory space in MB for $R_A$: Clear registers, T and H, and the flag $F_Q$ of Q and BR(i), $0 \leq i \leq p-1$, and $D \leftarrow 0$. (Register BR(i) is used to hold an address of ith module and D is used to hold the (identification) number of the module. Initially, BR(i) points to the starting address of ith module, $0 \leq i \leq p-1$ and D points to the first module.) Scan the relation B and output the sequence of $Y_b$-components $y_b$'s of $R_B$ to IP. The IP accepts each $y_b$ and deposits it into Q. The $y_b$'s in Q are then fetched and used to test if the corresponding bits in rA and rB are set or not.

<u>Cases</u>: (i) if $rA(y_b) = 0$, i.e., the $y_b$ does not appear in the join column of $R_A$, then ignore the component $y_b$ .

(ii) if $rA(y_b) = 1$ and $rB(y_b) = 0$, i.e., the $y_b$ is first encountered, then $rB(y_b) \leftarrow 1$ and allocate memory space in MB for storing $R_A$ tuples with $X_a = y_b$ .

The setting of $rB(y_b)$ will prevent the subsequent incoming $y_b$ from re-allocating memory space in MB for those $R_A$ tuples having $X_a = y_b$ . The memory allocation is done as follows:

(a) $T \leftarrow r(y_b)$, i.e., the value of word $r(y_b)$ is saved in T-register, which is a temporary register.

(b) $r'(y_b) \leftarrow D$ and $r''(y_b) \leftarrow BR([D])$, where [D] is used to index one of BR(k), $0 \leq k \leq p-1$. Remember that r may be regarded as consisting of r' and r''.)

(c) $MB([r(y_b)]] = MB([r'(y_b)]\cdot[r''(y_b)]]) \leftarrow$ 'mark', where 'mark' is a special code used to mark the starting location of a block of tuples and '·' denotes as concatenation.

(d) $BR([D]) \leftarrow BR([D]) + (T + 1)$ and $D \leftarrow (D + 1)$ module p. The former statement indicates that if there is any memory allocation assigned to the module specified by D, the allocation will start at the new 'logical' location BR([D]). (Each 'logical' location can hold a tuple.) We add one extra word for each allocation to store the 'mark'. The later one indicates that next allocation will be assigned to the next module.

The above three statements allocate a block of (T + 1) 'logical' words in the module specified by D-register (before updating) for storing $R_A$ tuples with $X_a = y_b$ . The condition $rA(y_b) = 1$ and $rB(y_b) = 1$ indicates that the block allocation for $R_A$ tuples with $X_a = y_b$ has been done.

(4) Output $R_A$ tuples and store the relevant tuples of the join in the allocated memory: Clear rA and registers, T and H, and the flag $F_Q$ of Q. Scan the relation A and output and store the sequence of $R_A$ tuples in Q. The tuples in Q are fetched and the $x_a$'s are extracted. The $x_a$'s are used as indices to address the corresponding rB bits. Each addressed rB bit is tested for being set or not.

Cases: (i) If $rB(x_a) = 0$, i.e., the $R_A$ tuple being held is irrelevant to the join since its join column value $x_a$ does not appear in the join column of $R_B$, then ignore the tuple.

(ii) If $rB(x_a) = 1$, then $r''(x_a) \leftarrow r''(x_a) + 1$ and $MB([r(x_a)]) \leftarrow$ 'the tuple being held'.

Notice that at the end of step (4), the address of the last 'logical' word of each block will be contained in the corresponding r word. This information is important to each server $S_i$ where new tuples are formed.

(5) Output $R_B$ tuples, deposit the relevant $R_B$ tuples of the join into the proper queues $Q_i$, and produce the concatenated tuples of the join: Clear T and H registers and the flag $F_Q$ of Q, and $T_i$ and $H_i$ registers and the $F_i$ flag for $0 \leq i \leq p-1$. Scane the $R_B$ tuples and output them to Q. The tuples in Q are fetched and their join column values $y_b$'s are extracted. The extracted $y_b$'s are used to test if the corresponding rB bits are set or not.

Cases: (i) If $rB(y_b) = 0$, i.e., the tuple being held is irrelevant to the join, ignore the tuple.

(ii) If $rB(y_b) = 1$, then fetch the $r''(y_b)$ and concatenate it to the $R_B$ tuple and deposit into the queue specified by the value in $r'(y_b)$. The $r''(y_b)$ holds an address pointing to the starting address of a block of $R_A$ tuples to which the $R_B$ tuples with

$Y_b = y_b$ will be concatenated.

Each server $S_i$ will start its joining of tuples in $Q_i$ and the tuples in M(i) once $Q_i$ is not empty (i.e., the contents of registers, $H_i$ and $T_i$, in $Q_i$ are not equal). After completing the join of two buckets, the next bucket-pair follows and so forth, until all the bucket-pairs have been processed. Logically, we can say that each $S_i$ produces the concatenated tuples of the join from two "sorted" such buckets. The concatenated tuples are stored in the corresponding buffers which are then either output to the MFC or stored back to the AP system for further processing.

So far, only a single join column is involved in the join operation. If a join on a composite column is concerned, additional codes for composite column values need to be assigned. Or, one can dynamically encode each composite column value with a unique value during operation. The later approach may lead to high implementation cost.

## 3.3.3. An Illustration Example

This section shows how the above algorithm works by means of an example. Consider the same database as given in Figure 2. As an example, we consider the equi-join of table SALES on column ITEM with table TYPE on column ITEM, though this join is not a type II explicit join. We will follow the steps of the above algorithm.

Step 1. Clear RAM, i.e., rA, rB, and r.

Step 2. Clear registers, T and H, and the flag $F_Q$ of Q. Assume that the sequence of ITEM-components of table SALES that are input to Q is <CAM>, <GEAR>, <CAM>, <NUT>, <CAM>, and <NUT>, as they appear in the SALES table of Figure 2. These components will be used as indexes to set the rA and update the corresponding counter words of r. At the end of this step, the rA would have the bit pattern (0, 1, 0, 1, 1, 0,...,0) and their corresponding values of the array r of words will be (0, 3, 0, 1, 2, 0,...0) (Figure 3(a)) - i.e., there are three SALES tuples with ITEM-component = <CAM>, one SALES tuple with ITEM-component = <GEAR> , and two SALES tuples with ITEM-component =<NUT>.

Step 3. Clear T, H, and $F_Q$ in Q, and BR(i), $0 \leq i \leq p-1$, and set D to 0. Assume that the sequence of ITEM-components of table TYPE input to Q is <BOLT>, <CAM>, <COG>, <GEAR>, <NUT>, and <SCREW>. These values are used as indexes to test the rA bits for being 1 or 0. Since rA(<BOLT>) = rA(0) = 0, ignore the BOLT . Since rA(<CAM>) = rA(1) = 1 and rB(<CAM>) = rB(1) = 0 (initially, rB is set to 0), set rB(1) = 1 and allocate memory space in MB for those SALES tuples with ITEM = <CAM>. Since r(<CAM>) = r(1) = 3, thus, 4 logical words must be allocated. The allocation will do the following:

(a) Save the contents of word r(<CAM>), now being 3, in T.

(b) Store the contents of D-register, now being 0, in r'(<CAM>) and the contents of BR([D]) = BR(0), now being 0, in r"(<CAM>). The word r(<CAM>) = r(1) now is a pointer word pointing to the starting address of the first module. (In fact, the setting of rB bits can be used to distinguish pointer words from counter words.)

(c) MB([r(<CAM>)]) = MB(0•0) ← 'mark'.

(d) (i) Increment BR(D) = BR(0) by 4 ( = T+1) so that if there is any memory allocation assigned to the first module, it will be allocated starting from the fifth logical word (i.e. logical address 4).
(ii) D ← (D+1) modulc P, now D indicating that next allocation, if any, will be assigned to the module next to the current one.

The third incoming value is <COG>. Since rA(<COG>) = rA(2) = 0, ignore the value. The same procedure is repeatedly applied to other values. At the end of this step, the bit array store rB and r will be (0, 1, 0, 1, 1, 0,...,0) and (0•0, 0•0, 0•0, 1•0, 2•0, 0•0,...) (Figure 3(b)), where r(1) = 1•0 = r'(1)

• r"(1) (i.e., concatenation) and the contents of all registers in CP are also shown in Figure 3(b).

Step 4. Clear rA and registers T and H, and the flag $F_Q$ of Q. Assume that the sequence of SALES tuples input to Q is the same as that of SALES tuples appearing in Figure 2. Any tuples with rB(x) = 1 (x being the ITEM-component of SALES) will be stored in the logical location in MB pointed by r(x). The first incoming tuple with rB(<CAM>) = 1 is stored in the logical location 1 of the first module M(1). (After this, the r"(1) has the value 1, which is initially set to 0 and incremented by 1 when a tuple is stored.) The second incoming tuple with rB(<GEAR>) = 1 is stored in the logical location 1 of M(2); the third tuple with rB(<CAM>) = 1 is stored in the logical location 2 of M(1); and so forth, until the sixth incoming tuple which is stored in the logical location 2 of M(3). At the end of this step, r"(1), r"(3), and r"(4) have the values 3, 1, and 2, respectively. Figure 3(c) shows the contents of RAM and the first three modules M(0), M(1), and M(2).

Step 5. Assume that the sequence of TYPE tuples input to Q is the same as that of TYPE tuples appearing in Figure 2. Any tuples with rB(y) = 0 (y being the ITEM-component of TYPE) are ignorant. Those tuples with rB(y) = 1 will be dispatched into the queues $Q_i (0 \leq i \leq p-1)$ determined by r'(y). They are concatenated to the contents of r"(y) before dispatching into the proper queues. The first incoming tuple is ignored since rB(<BOLT>) = rB(0) = 0; the second one concatenated to the contents of r"(<CAM>) = r"(1) = 3 is dispatched into the first queue $Q_0$ since r'(<CAM>) = r'(1) = 0; the third tuple is ignored; the fourth one concatenated to the contents of r"(<GEAR>) = r"(3) = 1 is dispatched into $Q_1$ since r'(<GEAR>) = r'(3) = 1; the fifth one concatenated to the contents of r"(<NUT>) = r"(4) = 2 is dispatched into $Q_2$ since r'(<NUT>) = r(4) = 2; the sixth tuple will be ignored. Since each tuple dispatched is associated with a pointer pointing to the last tuple of of the block to which the dispatched tuple is concatenated, each server $S_i$ thus can produce the concatenated tuples of the join from each TYPE tuple in $Q_i$ and the block of SALES tuples in M(i), without memory addressing contention problem.

## 4. ANALYSIS

The system performance is substantially influenced by hardware parameters such as the size of the RAM and the memory bank MB, the number of servers, the length of server queues, etc., especially when the type II explicit join is concerned. Our analysis assumes that the RAM and MB are large enough and hence concentrate on that how the number of servers and the length of server queues affect the hardware performance in computing the type II explicit join.

The analysis which follows will assume that the data transfer rate from the AP system to the hardware is high enough, thereby keeping the central processor (CP) busy. The analysis is divided into two aspects: one is to, given an application, determine the number of servers required and the length of their associated queues, without blocking the data deposited to the array of servers. (In our approach, an application is characterized in terms of the number of tuples of the relations being joined, the number of attributes in a relation, and the number of distinct values of the join columns.) The other is, given the number of servers, to determine how good the hardware performance is under different applications.

Figure 4 shows the overall structure of the step (S) of the type II explicit join. There are two stages: one is the CP and the other is the array of servers each associated with a queue. In stage 1, the average service time for CP to process one tuple is $t_{CP}$. $C(R_2)$ is the number of tuples of relation $R_2$. $S_{R2}$ is the selectivity of $R_2$ (the second relation to be joined), i.e., the ratio $C(R_2')/C(R_2)$, where $R_2'$ is a subrelation of $R_2$, in which tuples are relevant to the join. Thus, in average, there will be one tuple depositing into stage 2 in every $t_{CP}/S_{R2}$ seconds. If the number of servers is $N_p$, the average service time for the $N_p$ servers to process one tuple will be $t_s/N_p$, where $t_s$ is the average service time for a single server to process one tuple.

The condition for the hardware to be efficient would be $t_s/N_p < t_{CP}/S_{R2}$, i.e.,

$$N_p > \frac{t_s}{t_{CP}/S_{R2}} \quad .$$

This leads to the formula of calculating the optimal number $0_{nos}$ of servers:

$$0_{nos} = \left\lceil \frac{t_s}{t_{CP}/S_{R2}} (1+w) \right\rceil \tag{4-1}$$

where ⌈ ⌉ is the ceiling function and $w$ is the waste factor whose typical value is 0.1.

## 4.1. Simulation Approaches

Four simulation approaches are discussed in the following analysis. A hardware simulator is developed on the PDP-11/70 which simulates the hardware down to the logic level. It is provided for the simulation purposes. Applications are generated using a random number generator for generating relations.

### (1) Approach 1

This approach is to determine $S_{R2}$ and $N_p$ under different applications. These two values are then used to calculate the optimal number $0_{nos}$ of servers for each application.

Through the time complexity analysis of the step (5) of the type II explicit join, we have formulas of $t_s$ and $t_{CP}$ as follows:

$$t_s = t_{iq} + t_{ram} + t_{lu} + S_{R2} \cdot [(t_{iq} + t_{sq}) \cdot DG_{R2} + t_{sq}] \tag{4-2}$$

$$t_{CP} = (DG_{R2} + 1) \cdot t_{sq} + N_p \cdot [t_{c0} \cdot (DG_{R1} + DG_{R2}) + t_{add} + (t_{mb} \cdot DG_{R1}) \tag{4.3}$$

where

| | |
|---|---|
| $t_{iq}$ : | time for accessing one attribute value in the input queue. |
| $t_{sq}$ : | time for accessing one attribute value in the server queue. |
| $t_{mb}$ : | time for accessing one attribute value in the memory bank. |
| $t_{ram}$: | time for accessing one bit/word in RAM. |
| $t_{add}$: | time for CP to perform one addition/subtraction. |
| $t_{lu}$ : | time for examining the contents of one rA/rB bit. |
| $t_{co}$ : | average time for a server to form a new tuple. |
| $DG_{R1}/DG_{R2}$ : | number of sttributes in $R_1/R_2$. |

In our simulation, we assume that the values of $t_{iq}$, $t_{sq}$, $t_{ram}$, and $t_{mb}$ are all 100 ns, $t_{add}$ and $t_{lu}$ 50 ns, and $t_{co}$ 100 ns. To make the analysis easier to accomplish, we consider an application with:

$$DG_{R1} = DG_{R2} = 10$$

$$D(R_1, R_2) = 100$$

$$C(R_1) = C(R_2) = C(R)$$

where D(R1, R2) is the number of distinct values of the join columns in relation R1/R2. Thus, we have

$$O_{nos} = \left\lceil \frac{1100 + N_p \cdot 3050}{250/S_{R2} + 2100} \cdot (1.1) \right\rceil . \tag{4.4}$$

where both $N_p$ and $S_{R2}$ are functions of C(R) and $D(R_1, R_2)$. More precisely, they are functions of the ratio $C(R)/D(R_1, R_2)$. From the results of simulation, we can estimate the values of $S_{R2}$ and $N_p$ for different applications, and using equation (4.4) to calculate the $O_{nos}$.

Figure 5(a) shows our simulation results with $D(R_1, R_2)$ fixed and varying C(R). It indicates that the log $O_{nos}$ is proportional to $\log[C(R)/D(R_1, R_2)]$, since increasing $C(R_1)/D(R_1, R_2)$ will no doubt increase the size of resulting relation of the join, and therefore need more servers to carry out the concatenated tuples of the join.

(2) Approach 2

In this approach, we fix $O_{nos}$ determined from Approach 1, and vary the length of queues to estimate the performance of system architecture. Let $t_5$ be the average service time required to serve one $R_2$ tuple in step (5) ($t_5 = T_5/C(R_2)$, where $T_5$ is the total time required in step (5) of the type II explicit join). We use $t_5/t_{CP}$ as the base of performance evaluation.

Figure 5(b) shows the results of this approach. It indicates that only a few units of tuple are required for server queues and the queue length itself is not a sensitive factor to different applications.

(3) Approach 3

In this approach, we fix the number ($N_p = 12$) of servers and the length ($l_{ser} = 4$) of server queue, to estimate the system performance under different applications. The base is still $t_5/t_{CP}$. The $O_{nos}$'s required in each application are also shown in Figure 5(c). We can see that the performance will greatly degrade (i.e., $t_5/t_{CP} > 1$) if an application has the condition that the $O_{nos}$ it requires is much greater than the given $N_p$ (= 12 in this case).

(4) Appooach 4

This approach is similar to approach 3, but we choose the applications that have the ratio of input arrival rate to the array of servers and output servive rate close to 1. The effect of the length of service queues on the system performance is studied. The results in Figure 5(d) shows that the length of service queues affects the system performance only when the ratio $(t_s/N_p)/(t_{CP}/S_{R2})$ is close to 1. The application with $\log[C(R)/D(R1,R2)]$ = 9 will satisfy this condition in our example ($N_p$=12). That is, the length of service queues will affect the system performance only when an application with the rate of input arrival rate and output service rate close to 1 is concerned. The result is consistent with the queueing theory.

## 5. SUMMARY AND CONCLUSION

We have shown how the RAM helps perform implicit joins and type I explicit joins. We have also shown how the hardware performs the type II explicit join. Through the use of the RAM, the joining of two relations in which their join columns satisfy the referential integrity rule can be implemented as effectively as the implicit join. Since a majority of explicit joins are performed via the relationship defined in this rule, the extention of single bit array stores to an RAM is essential.

The analysis reveals that in the case where the overall computing speed of servers is greater than that of CP, the time required in the type II join operation is linear in the cardinality of the two relations being joined, independent of the cardinality of the resulting relation of the join. This independence is the result of the use of the array of servers.

We have proposed an algorithm for calculating the optimal number of servers and the length of the server queues. The simulation shows that the optimal number of servers is linear with respect to the ratio of C(R) to $D(R_1,R_2)$ and, given a fixed number of servers; one can determine how good the hardware performance is under different applications.

This hardware provides powerful join capabilities to AP systems, especially when a join-dominating database application is concerned. We believe that it can be adapted to the current VLSI technology and has the important feature of being applicable with little or no modification to currently proposed AP hardware. The new version of an AP system allows that a search on a single relation is performed by the AP hardware, while a joining of two relations is carried out by the extented hardware.

## 6. REFERENCES

[1] Bobb, E., "Implementing a Relational Database by Means of Specialized Hardware," ACM TODS, Vol.4, 1, March 1979, pp.1-29.

[2] Banerjee, J., and Hsiao, D. K., "DBC — A Database Computer for Very Large Databases," IEEE Trans. on Computers, Vol.C-28, 3, 1979.

[3] Chang, H., "On Bubble Memories and Relational Data Base," Proc. 4th Int'l Conf. on VLDB, West Berlin, 1978, pp.207-229.

[4] Chen, T. C., Lum, V. W., and Tung, C., "The Rebound Sorter : An Efficient Sort Engine for Large Files," Proc. 4th Int'l Conf. on VLDB, West Berlin, 1978, pp.312-315.

[5] Date, C. J., An Introduction to Database Systems, Addison-Wesley, Reading, Mass., Third ed., 1981.

[6] Edelberg, M., and Schissler, L. R., "Intelligent Memory," Proc. 1976 NCC, Vol.45, AFIPS Press, Montuale, N. J., pp.691-701.

[7] Hong, Y. C., and Su, S. Y. W., "Associative Hardware and Software Techniques for Integrity Control," ACM TODS, Vol.6, 3, Sept. 1981, pp.416-440.

[8] Hong, Y. C., and Su, S. Y. W.,"A Mechanism for Database Protection in Cellar-Logic Devices," IEEE Trans. Software Engineering, Nov. 1982.

[9] McGregor, D. R., Thomson, R. G., and Dawson, W. N., "High Performance for Database Systems," Systems for Large Databases, North-Holland Publishing Co., 1976, pp.103-116.

[10] Menon, M. J., and Hsiao, D. K., "Design and Analysis of a Relational Operation for VLSI," Proc. 7th VLDB, Paris, France, 1981, pp 44-55.

[11] Lin, C. S., Smith, D. C. P., and Smith, J. M., "The Design of a Rotating Associative Memory for Relational Database Applications," ACM TODS, Vol.1, 1, March 1976, pp.53-65.

[12] Ozkarahan, E. A., Schuster, S. A., and Smith, K. C., "RAP — an Associative Processor for Database Management," Proc. 1975 NCC, Vol.44, AFIPS Press, Montvale, N. J., pp.379-387.

[13] Shaw, D., "A Relational Database Machine Architecture," Proc. 5th Annual Workshop on Computer Architecture for Non-Numeric Processing, Pacific Grore Ca., March 1980.

[14] Smith, D. C. P., and Smith, J. M., "Relational Database Machines," IEEE Computers, Vol.12, 3, March 1979, pp.28-37.

[15] Su, S. Y. W., "On Logic-Per-Track Devices : Concepts and Applications," IEEE Computers, Vol.12, 3, March 1979, pp.11-25.

[16] Su, S. Y. W., and Lipovski, G. J., "CASSM: A Cellular System for Very Large Databases," Proc. Int'l Conf. on VLDB, Sept. 1975, pp.456-472.

[17] Su, S. Y. W., Nguyen, L. H., Eman, A., and Lipovski, G. J., "The Architectural Features and Implementation Techniques of the Multicell CASSM," IEEE Trans. on Computers, Vol. C-26, 6, June 1979, pp.430-445.

[18] Tanaka, Y., Nozaka, Y., and Masuyama, A., "Pipeline Searching and Sorting Modules as Components of a Data Flow Database Computer," Proceedings of IFIP Congress 80, pp.427-432.

[19] Todd, Stephen, "Hardware Design for High Level Databases," IBM United Kingdom Scientific Center, Peterlee, TN 49.

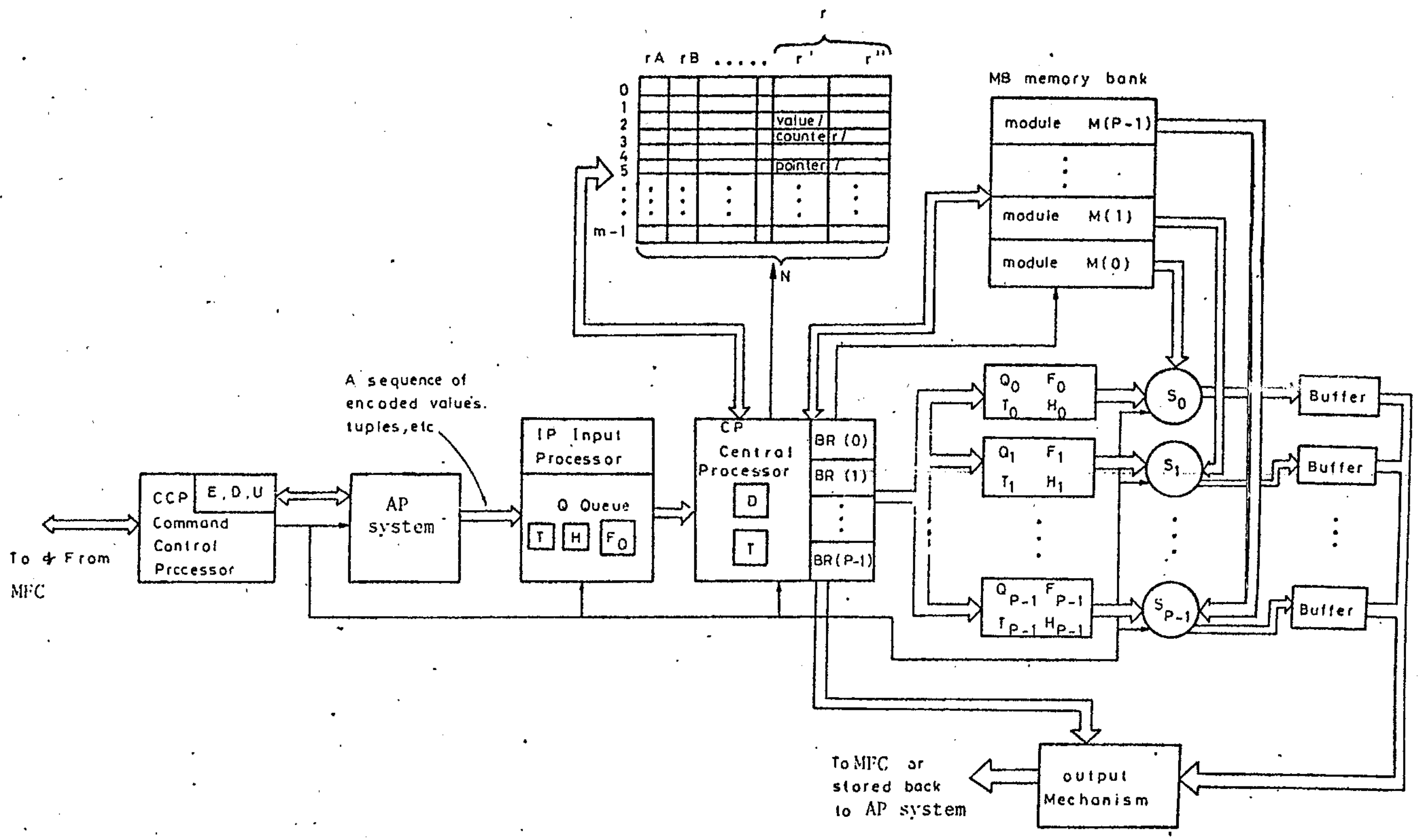


Figure 1. Hardware Architecture

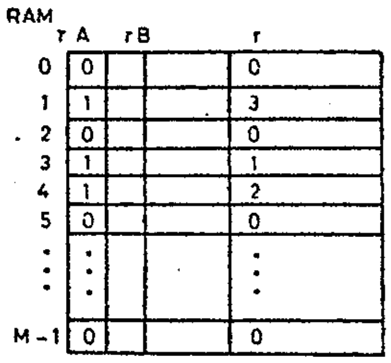
SALES

| DEPT | ITEM |
|---|---|
| <D1> | <CAM> |
| <D1> | <GEAR> |
| <D5> | <CAM> |
| <D5> | <NUT> |
| <D8> | <CAM> |
| <D10> | <NUT> |

TYPE

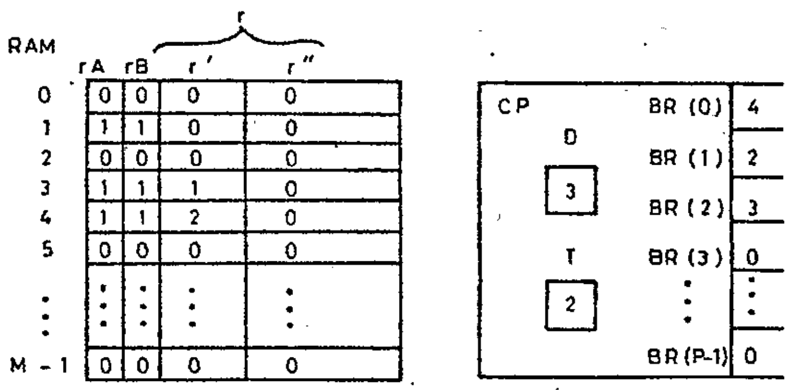
| ITEM | COLOR | PRICE |
|---|---|---|
| <BOLT> | <GREEN> | <5p> |
| <CAM> | <RED> | <2p> |
| <COG> | <RED> | <4p> |
| <GEAR> | <GREEN> | <4p> |
| <NUT> | <BLACK> | <8p> |
| <SCREW> | <YELLOW> | <7p> |

Figure 2. A Simplified Database With Two Tables SALES And TYPE Linked By ITEM.


RAM

| | rA | rB | | r |
|---|---|---|---|---|
| 0 | 0 | | | 0 |
| 1 | 1 | | | 3 |
| 2 | 0 | | | 0 |
| 3 | 1 | | | 1 |
| 4 | 1 | | | 2 |
| 5 | 0 | | | 0 |
| ⋮ | ⋮ | | | ⋮ |
| M-1 | 0 | | | 0 |

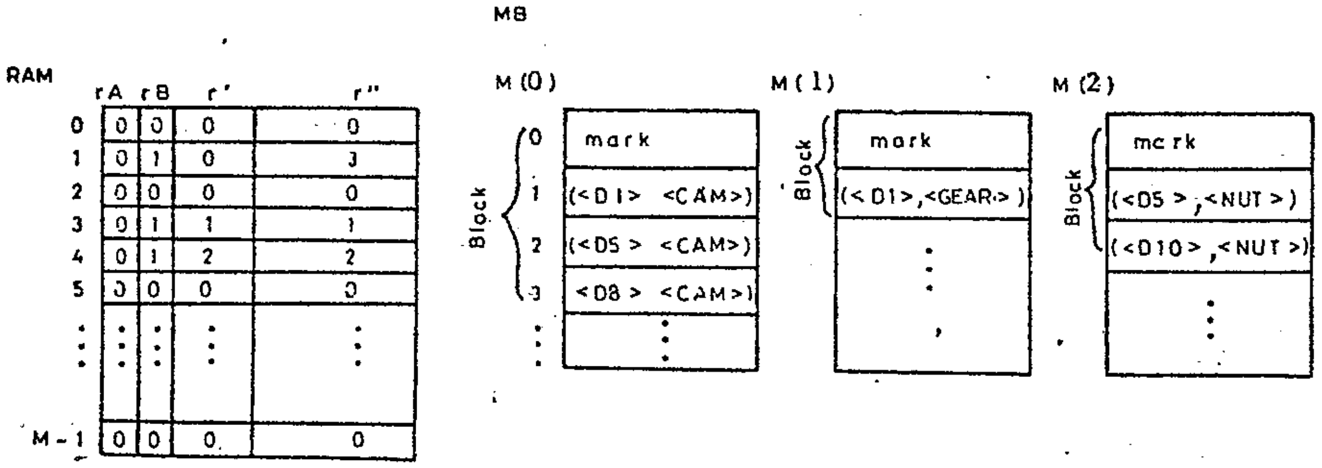
Figure 3(a)


RAM

| | rA | rB | r' | r'' |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 |
| 3 | 1 | 1 | 1 | 0 |
| 4 | 1 | 1 | 2 | 0 |
| 5 | 0 | 0 | 0 | 0 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| M - 1 | 0 | 0 | 0 | 0 |

| | |
|---|---|
| BR (0) | 4 |
| BR (1) | 2 |
| BR (2) | 3 |
| BR (3) | 0 |
| ⋮ | ⋮ |
| BR (P-1) | 0 |

Figure 3(b)


RAM

| | rA | rB | r' | r'' |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 3 |
| 2 | 0 | 0 | 0 | 0 |
| 3 | 0 | 1 | 1 | 1 |
| 4 | 0 | 1 | 2 | 2 |
| 5 | 0 | 0 | 0 | 0 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| M - 1 | 0 | 0 | 0 | 0 |

MB

M (0)

| | |
|---|---|
| 0 | mark |
| 1 | (<D1> <CAM>) |
| 2 | (<D5> <CAM>) |
| 3 | <D8> <CAM> |
| ⋮ | ⋮ |

M (1)

| |
|---|
| mark |
| (<D1>,<GEAR>) |
| ⋮ |

M (2)

| |
|---|
| mark |
| (<D5>,<NUT>) |
| (<D10>,<NUT>) |
| ⋮ |

Figure 3(c)

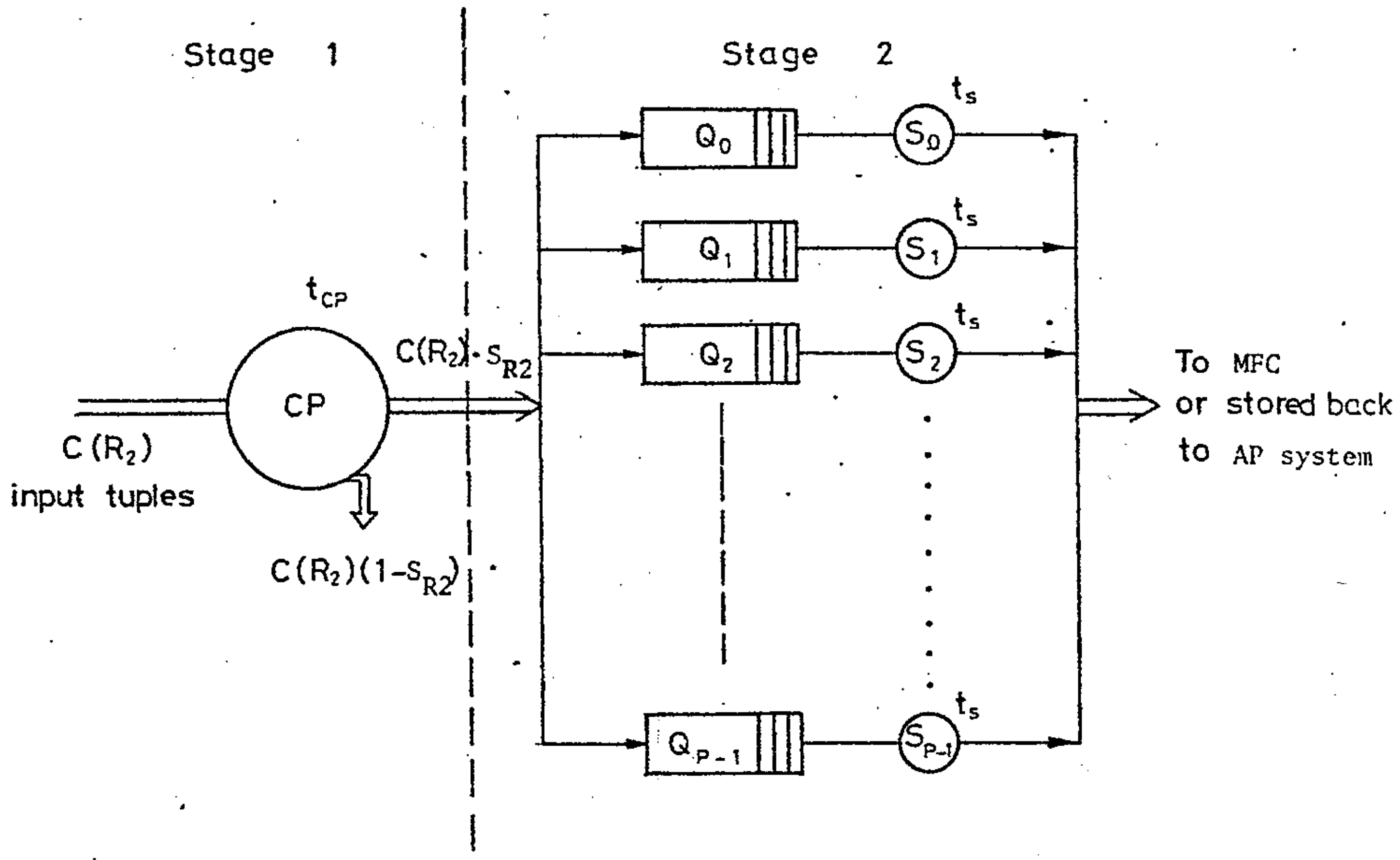
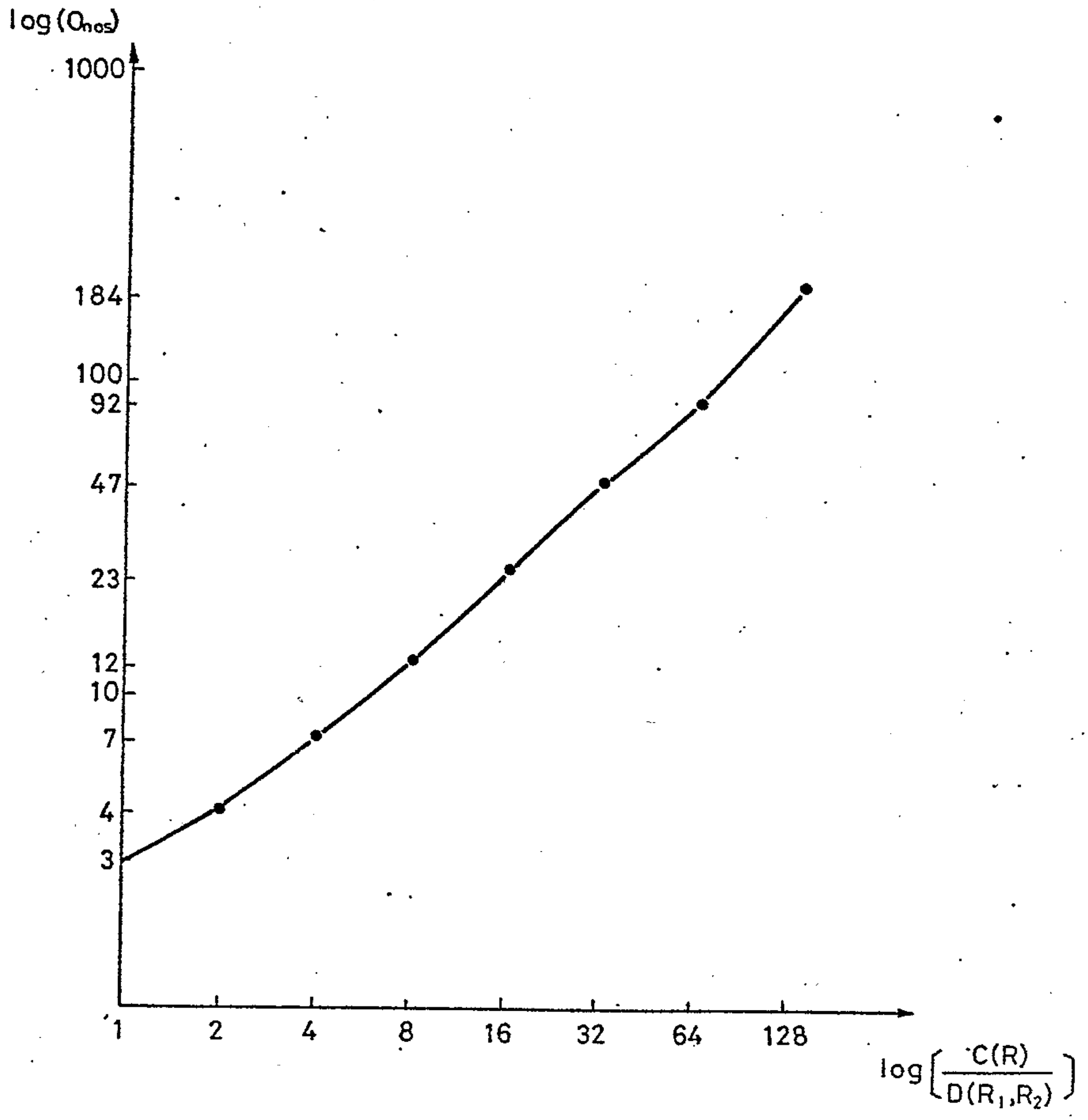

Stage 1
Stage 2
$t_{CP}$
CP
$C(R_2)$
input tuples
$C(R_2) \cdot S_{R2}$
$C(R_2)(1-S_{R2})$
$Q_0$
$Q_1$
$Q_2$
$Q_{P-1}$
$t_s$
$S_0$
$S_1$
$S_2$
$S_{P-1}$
To MFC
or stored back
to AP system

*Figure 4*



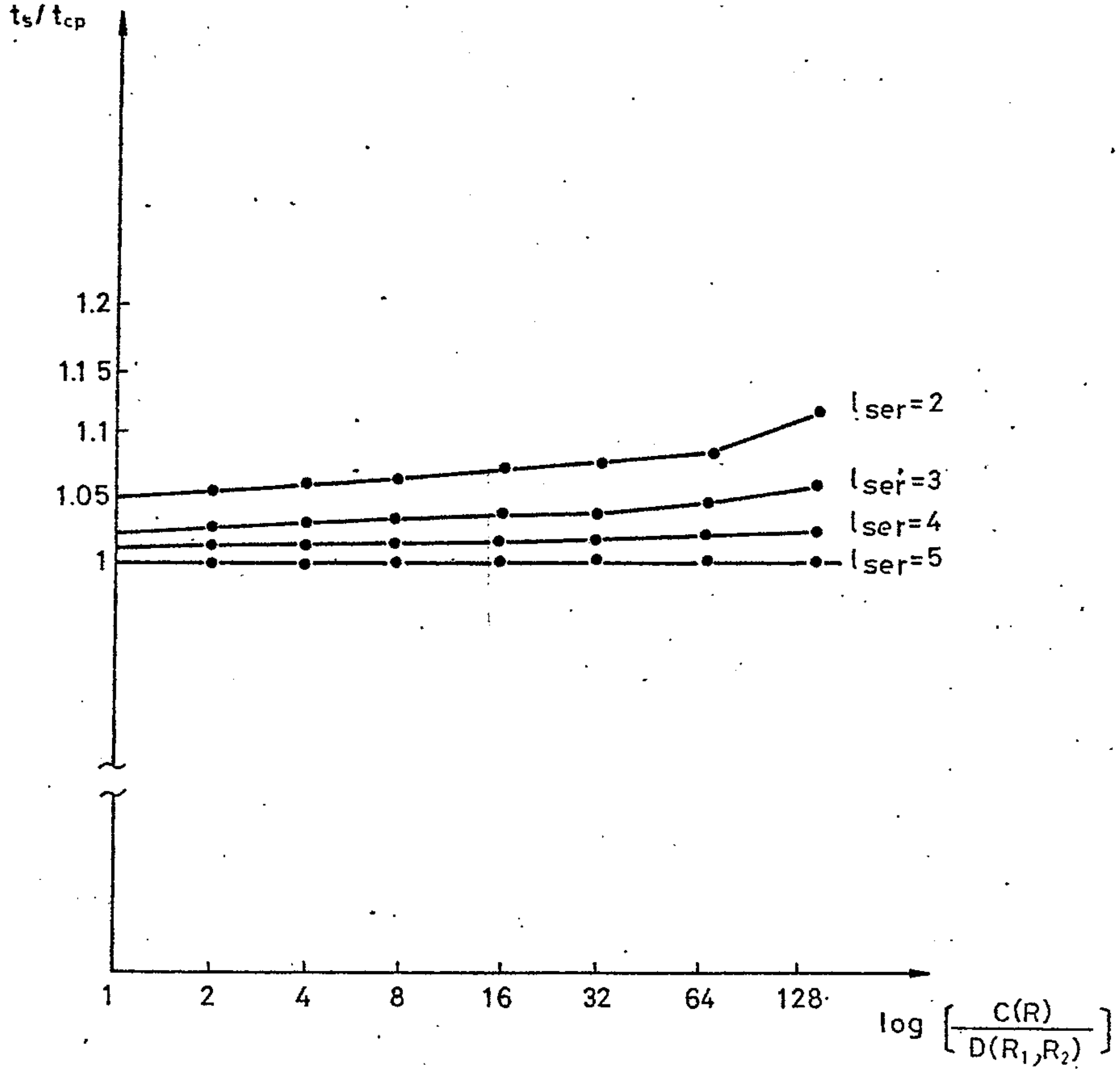
$DG_1 = DG_2 = 10$

$D(R_1, R_2) = 100$

$C(R_1) = C(R_2) = C(R) = 100, 200, 400, 800, 1600$

$3200, 6400, 12800$

Figure 5(a)

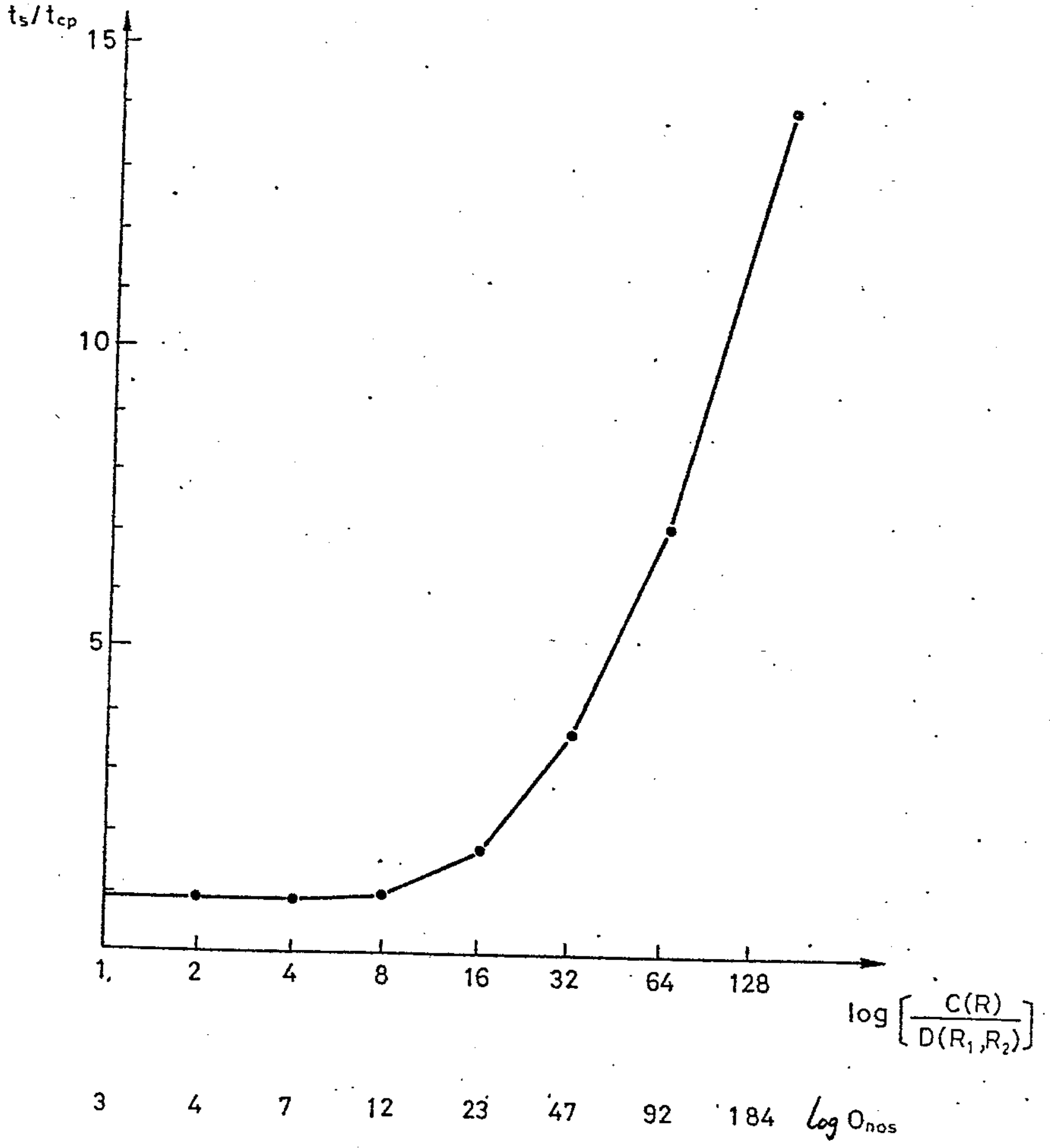


$DG_1 = DG_2 = 10$

$C(R_1) = C(R_2) = C(R)$

$D(R_1, R_2) = 100$

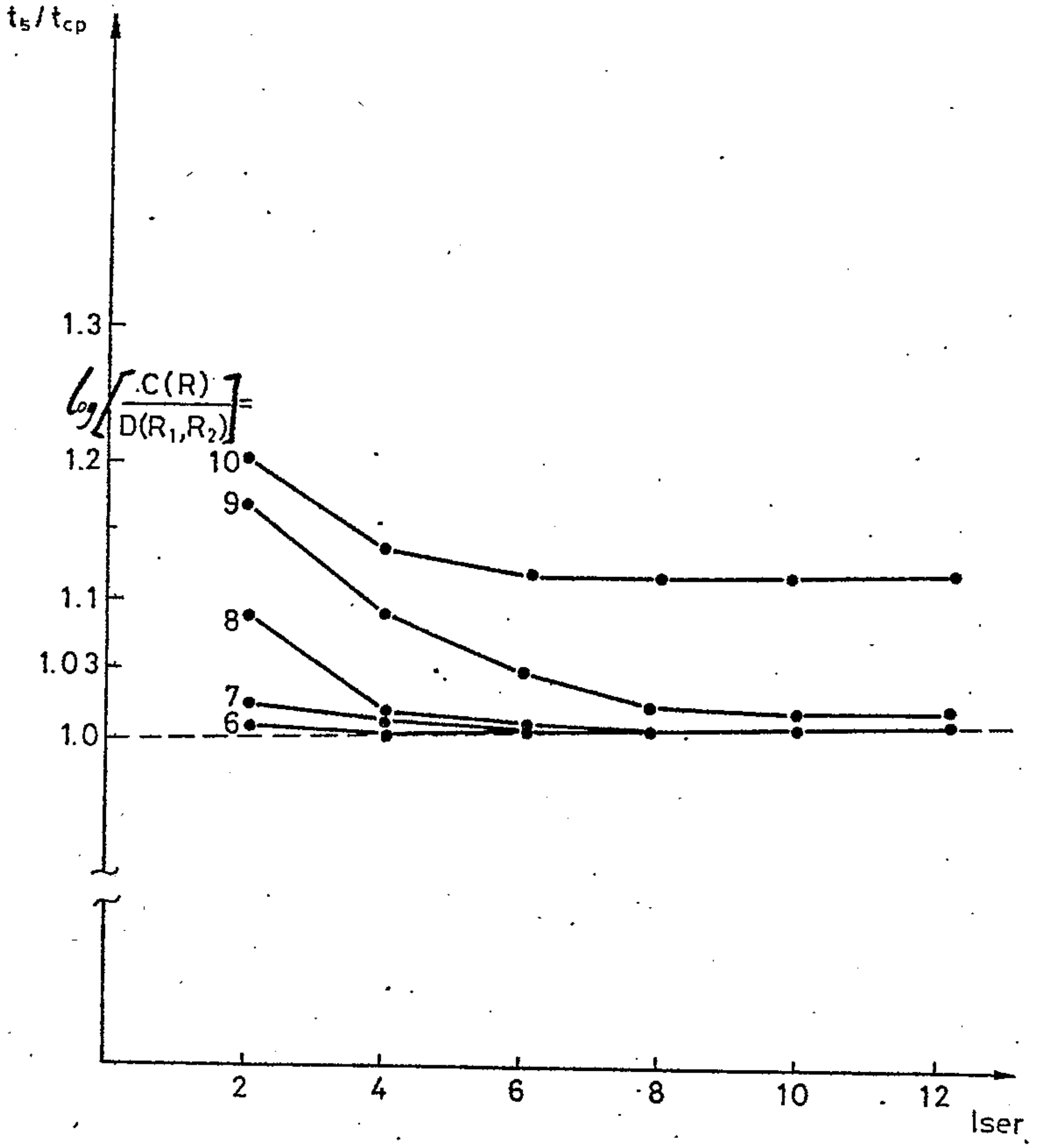
Figure 5 (b)



$\log {}^{N}P = 12$

Iser = 4 (in tuple unit)

Figure 5(c)



$$\log\left[\frac{C(R)}{D(R_1,R_2)}\right] = \quad 6 \quad , \quad 7 \quad , \quad 8 \quad , \quad 9 \quad , \quad 10$$

$$r_a = \frac{t_s / N_p}{t_{cp} / s_{R2}} = 0.687 \ , \ 0.796 \ , \ 0.904 \ , \ 1.012 \ , \ 1.120$$

$$\log N_P = 12$$

Figure 5(d)