Bull. Nov. Comp. Center, Comp. Science, 2 (1993), 45–62
© 1993 NCC Publisher

Investigation of some hardware accelerators for relational algebra operations

A.Sh. Nepomniaschaya and Ya.I. Fet

This paper is devoted to the application of specialized processors for speeding-up the realization of relational algebra operations on fine-grain SIMD computers. A short description of two processors is presented. By means of a special high level language STAR, the algorithms of implementation of relational algebra operations in a STARAN-like computer model are described. Then the representation of the same algorithms is considered for the same model with attached processors. The comparison is made, corroborating the efficiency of the proposed co-processors.

1. Introduction

Fine-grain SIMD systems with bit-serial (vertical) processing and simple single-bit PEs have been the subject of intensive research in recent years [1]. To this class of architectures belong the well-known systems STARAN, DAP, MPP, CM. We call such multiprocessors Vertical Processing Systems, VPS [2].

These systems provide very high performance due to the concurrent operation of a large number of PEs. Nevertheless, an additional speed-up can be attained in VPS for some applications using co-processors, designed with allowance for a specific nature of problems under consideration.

One of the most important fields of VPS application is the non-numeric processing, i.e., large data bases, knowledge bases, expert systems, and other problems of artificial intelligence. In solving these problems, a set of relational algebra operations is usually used as an operational basis. Implementations of the relational algebra in conventional associative processors and specialized parallel processors were discussed in [3–5] and other publications. Note that the relational algebra also found important applications in other advanced data and knowledge models (see, for example, [6]).

In this paper, two specialized processors are considered (called "λ-compressor" and "ω-matrix") intended for speeding-up the realization of relational algebra operations in VPS's. The use of these processors is demonstrated in the environment of a STARAN-like associative computer. In order to evaluate the speed-up, two models are considered: the usual model of an associative array processor and a modified model of the same processor joined with the λ-compressor and the ω-matrix.

In [7] J.L. Potter has proposed the associative computing language (ASC) for implementing associative and semantic networks on massively parallel SIMD computers. It is a convenient high-level language for the specification of AI algorithms for such computers. However, to write different algorithms for VPS saying "how to do", special tools were considered in [8–10]. Nevertheless, for description and evaluation of new hardware devices it would be useful to have at hand a high-level language allowing complete modeling of parallel associative processing. To this end the language STAR was proposed in [11]. It resembles Pascal, but has special data types and corresponding operations for them allowing one to simulate easily the run of associative architecture.

The paper is constructed in the following way.

In Section 2 a model of a STARAN-like associative machine is described. Section 3 contains a short review of the language STAR and Section 4 gives a formal description of relational algebra implementation in the STARAN-like model. In Section 5 the necessary information is given concerning the processors λ and ω. In Section 6 some additional language constructions are introduced into STAR allowing the description of specialized hardware. Then, the algorithms of Section 4 are rewritten taking into account the specialized devices. Eventually in the Conclusion the comparison of the results from Sections 4 and 6, corroborating the efficiency of the proposed co-processors, is adduced.

2. Model of associative parallel machine

We consider the associative machine STARAN as a basic architecture. To describe our model, we use an abstract STAR-machine of SIMD-type with vertical data processing. It consists of the following components:

1) a sequential common control unit (CU), where programs and scalar constants are stored;
2) k associative processors ($k \leq 32$), each consisting of m single-digit processor elements ($m = 256$);
3) k matrix memory modules, where the i-th module is connected with the i-th processor ($1 \leq i \leq k$).

Each memory module consists of r blocks ($r \leq 16$) and each block consists of m words by m bits. In any block the rows are numbered from top to bottom and the columns are numbered from left to right. A row (word) or a column (slice) may be accessed equally easily. The data are viewed as a two-dimensional array written in the binary code. Each array element occupies an individual row and all elements have the same length. The data array is divided into parts each of m rows. They are loaded into the module blocks so that each part is stored in a block and different parts are stored in different modules. In the CU a rendition table should be located allowing one to associate with each array identifier the number of columns and parts in the partitioning.

Each associative processor can be represented as h vertical registers each of m bits. The bit columns of the data array are stored in the registers which perform the necessary Boolean operations, record search results and ensure the word selection capability. The STAR-machine processor has a sufficient number of vertical registers ($h \geq 3$) to store intermediate results of data processing without using the module memory.

In this paper we consider a STAR-machine with one associative processor ($k = 1$). In this case all parts of the data array are loaded into one module. The CU decodes program instructions and causes the processor to execute them. The processor performs vertical data processing for each block in turn.

3. Review of the language STAR

We consider only those STAR constructions [11] which are necessary for the description of relational algebra operations. To simulate data processing in a memory block, the following data types are used: *integer, boolean, word, slice, table* and *array*. Data types are introduced in the same manner as in Pascal. Constants for the types *slice* and *word* are represented as an ordered sequence of symbols 'zero' and 'one' enclosed within single quotation marks (apostrophes). Note that the types *slice* and *word* are introduced for the bit column access and the bit row access, respectively.

Let M be a variable of the type *array*. Then M is a structure of a fixed number of components, all being of the same type *integer*. Let T be a variable of the type *table*. Then T is associated with the matrix T of k columns where $k \leq 256$.

By analogy with [4] we assume that any matrix T has a unique bit-slice (called *workfield*) TWF indicating by '1' those rows which belong to T.

3.1. Operations, predicates and standard functions for slices

Any variable of the type *slice* consists of 256 components which belong to $\{0,1\}$.

Let X and Y be variables of the type *slice*, i be a variable of the type *integer*. We consider that ONE-component and ZERO-component of a slice denote its component with the value '1' and '0', respectively.

Define the following operations:

CLR(Y) sets all components of Y to ZERO;

SET(Y) sets all components of Y to ONE;

$Y(i)$ selects the i-th component of Y;

NUMB(Y) yields the number i of all ONE-components of Y, $i \geq 0$;

FND(Y) yields the ordinal number i of the first ONE-component of Y, $i \geq 0$;

STEP(Y) yields the same result as the operation FND(Y) and then resets the first ONE-component of Y. If the slice Y has no ONE-components, it does not change;

PRESS(X, Y) deletes from the slice X those components which correspond to positions '0' in the slice Y and then compresses the contents of X. If there are k ZERO-components in the slice Y, then the last k components of the slice X will be '0'.

The following logical operations are executed simultaneously by all corresponding components of X and Y and introduced in the obvious way:

$X \wedge Y$ is conjunction, $X \vee Y$ is disjunction, $\neg X$ is negation.

Other logical operations are constructed from these operations by means of superposition. Let us agree that $X \oplus Y$ denotes the operation exclusive 'OR'.

There are the following three predicates for slices:

ONE(Y) yields *true* if and only if the slice Y consists completely of ONE-components;

ZERO(Y) yields *true* if and only if the slice Y consists completely of ZERO-components;

SOME(Y) yields *true* if and only if the slice Y has at least one component with value '1'.

We use the following function Shift(Y, k) in which k is a variable of the type *integer*. This function moves the contents of Y placing the component from position N to position $N + k$ $(N \geq 1)$ and setting ZERO-components from the first through the k-th positions, inclusive. If $k = 0$, then the contents of Y does not change.

3.2. Operations and standard functions for words and matrices

Let w be a variable of the type *word*, i be a variable of the type *integer*. We use the following three operations:

$\#w$ yields the length of w $(\#w \leq 256)$;

$w(i)$ yields the i-th component of w;

dis(w) yields the disjunction of all components of w.

Let T be a variable of the type *table*, and i,j,k be variables of the type *integer*. We use the following matrix operations :

$T(i)$ yields the i-th row in the matrix T;

Col(j, T) yields the j-th column in the matrix T;

$T[k]$ yields the k-th part of the matrix T $(1 \leq k \leq 16)$. This operation is used when the matrix T has more than 256 rows.

The function Size(T) yields the number of columns in T;

The function Row(T) yields cardinality (the number of rows) of T.

Remark. It should be noted that statements of STAR resemble those of Pascal [12].

4. Algorithms for relational algebra operations

A relational database model is defined as in [13]. Let D_i be a domain, $i = 1, 2, \ldots, k$. The relation R is considered as a subset of the Cartesian product $D_1 \times D_2 \times \ldots \times D_k$. An element of R is called *tuple* and has the form $v = (v_1, v_2, \ldots, v_k)$, where $v_i \in D_i$. Let A_i be a name of the domain D_i which is called *the attribute*. Let $R(A_1, A_2, \ldots, A_k)$ denote *a scheme* of the relation R.

Any relation is represented as a matrix (table) in the memory module and each its tuple is allocated to one memory word. Therefore the values of attributes occupy the vertical fields in the matrix. Note that any relation consists of different tuples. If the relation R consists of several parts, then each part of R uses the same workfield bit-slice RWF in the module.

For simplicity we assume that different relations have different attributes. Therefore we can refer to any domain of the considered relation by using only the corresponding attribute. In general, if different relations use a common attribute, then the reference to the corresponding domain has the following form:

<relation name> . <attribute>

We introduce the following notation being used for algorithm analysis. Let ALG be an algorithm applied to the matrix T. Denote by $N(ALG)$ the access number to the parallel memory during the execution of ALG. Following [8], we assume that definition of existence of responder in the parallel memory needs no additional time.

4.1. Auxiliary procedures

In this section we consider a group of auxiliary procedures which will be used later. Some procedures will be considered in detail, but the other ones – only informally explained. Note that we borrow some auxiliary procedure names from [5].

First we consider the procedure MATCH which tests a word v for the membership in the relation D.

```
proc MATCH($D$:table; DWF:slice; $v$:word; var $M$:slice);
label 1; var $i, k$: integer; $X$: slice;
begin $k$:= size($D$); $M$:=DWF;
   for i:= 1 to $k$ do
      begin $X$:= col($i, D$); if $v(i) ='1'$ then $M := M \wedge X$
         else $M := M \wedge \neg X$; if ZERO($M$) then goto 1
      end;
1:end
```

A detailed explanation may be appropriate here. Note that the relation D and its bit-slice DWF are loaded in the parallel memory. Since the vertical processing is executed in the associative processor, it is necessary to have at least two variables of the type *slice*. The variable X is used to store the current column of D which is operated on and the variable M is used as the resulting bit-slice. At the start M has the same contents as DWF since the seach will be executed among those rows of D which correspond to the positions with '1' in the slice DWF. Let us call such rows of D by *selected* rows. At any j-th step of the algorithm $(1 \leq j \leq \#v)$

we will mark by '1' in the slice M the positions of those selected rows (tuples) of D which have the first j symbols of v as their initial part. This algorithm terminates earlier if there exists such a step $h < k$ in which the slice M has only ZERO-components.

Consider the procedure COMPACT.

```
proc COMPACT($D$:table; $Y$:slice; var $H$:table);
var $i, j$:integer; $Z$:slice; $w$:word;
begin $j := 0$; $Z := Y$;
  while SOME($Z$) do
    begin $i$:=STEP($Z$); $j := j + 1$; $w$:=$D(i)$; $H(j)$:=$w$
    end;
end
```

The procedure COMPACT constructs the matrix H consisting of those rows $D(i)$ of D, for which the i-th component of the slice Y is '1', i.e., $Y(i) =$ '1'.

Let us informally describe the following four procedures for which there exist simple algorithms of vertical processing.

```
proc CLEAR(var $D$:table);
proc PUSH($D$:table; $r$:integer; var: $H$:table);
proc WCOPY($w$:word; $k, j$:integer; var $D$:table);
proc TCOPY($D$:table; DWF:slice; $k$:integer; var $H$:table);
```

The procedure CLEAR sets ZERO-components in each column of D. The procedure PUSH shifts the contents of the matrix D by r positions down. The procedure WCOPY writes k copies of the word w beginning with the j-th row of the matrix D. We assume that $\#w =$ size(D). The procedure TCOPY constructs k copies of each column of the matrix D by means of the internal cycle and using an additional slice.

4.2. Relational algebra operations

In this section we consider a group of relational algebra operations for which a special hardware support will be described.

First we consider the operation Intersection for which the resulting relation is a subset of one of its argument relations. Therefore we will use a bit-slice to indicate the resulting relation tuples. This operation has two argument relations. Its resulting relation consists of those tuples which belong to both argument relations.

In [14] a simple algorithm for Intersection was considered. For each row w from the second relation (R) it successively determines all occurrences of w in the first relation (T) by using the algorithm from the procedure MATCH. Here we examine a vertical algorithm for the operation Intersection by generalizing the algorithm used in the procedure MATCH. Explain the main idea of this generalization. We determine synchronous occurrences of all rows from the relation R in the relation T. To this end a variable Y_k of the type *slice* is used for each row w_k of the relation R. At each i-th step of the computation ($i \leq \#w$) the variable Y_k stores positions of those rows from the relation T which have the first i symbols of w_k as their initial part. If w_k belongs to the relation T, then there is a unique j such that $Y_k(j) = '1'$. Note that the variables Y_k form an auxiliary matrix which is used to obtain the operation Intersection result.

To consider the mentioned above vertical algorithm we use the following three auxiliary procedures.

```
proc INIT($Y$:slice; var $S$:table);
var $i, k$:integer;
begin $k$:=size($S$);
  for $i := 1$ to $k$ do col($i, S$) := $Y$
end
```

The procedure INIT puts the contents of the slice Y into all columns of the matrix S.

```
proc DIS($S$:table; var $Z$:slice);
var $j, r$:integer; $w$:word;
begin $r$:=row($S$);
  for $j := 1$ to $r$ do
    begin $w := S(j)$; if ZERO($w$) then $Z(j) :='0'$
      else $Z(j) :='1'$
    end;
end
```

For each row w of the matrix S the procedure DIS fulfils the disjunction of its components.

Consider the third auxiliary procedure LINE in which variables A and B are used for selecting the i-th column in the matrices T and R, respectively, and the variable C is used for the control column RWF of the matrix R in the parallel memory of the STAR-machine. Notice that later on this procedure will be used after the procedure INIT(Y, S).

```
proc LINE($A, B, C$:slice; var $S$:table);
var $i, r$:integer; $F$:slice;
```

```
begin $r$:=NUMB($C$);
* Note that NUMB($C$)=row($R$) *
  for $i := 1$ to $r$ do
    begin $F$:=col($i,S$); if $B(i) =' 1'$ then
      col$(i,S) := F \wedge A$ else col$(i,S) := F \wedge \neg A$
    end;
end
```

It can be verified that

$$N(\text{INIT})=\text{size}(S),\ N(\text{DIS})=2\cdot\text{row}(S) \text{ and } N(\text{LINE})=3\cdot\text{row}(S)+1.$$

Now consider the vertical algorithm for the operation Intersection which uses the procedures INIT, DIS and LINE.

```
proc INTERSV($T, R$:table; TWF,RWF:slice; var $Z$:slice);
var $S$:table; $X, Y, M$:slice; $j, k$:integer;
begin $M$:=RWF; $k$:=size($R$); size($S$):=row($R$);
* Note that size($R$)=size($T$) *
  row($S$):=row($T$); INIT(TWF,$S$);
  for $j := 1$ to $k$ do
    begin $X$:=col($j,T$); $Y$:=col($j,R$); LINE($X,Y,M,S$)
    end;
    DIS($S,Z$)
end
```

It can be easily calculated that

$$N(\text{INTERSV})=1+\text{row}(R)+2\cdot\text{row}(T)+3\cdot\text{size}(R)\cdot(1+\text{row}(T)).$$

For the procedure INTERS from [14] it is not difficult to obtain the following estimation:

$$N(\text{INTERS})=3+(3+2\cdot\text{size}(R))\cdot\text{row}(R).$$

Without loss of generality we can assume that row(R) $\leq$row(T). Therefore N(INTERS)$<$ N(INTERSV). However, the procedure INTERSV is useful in this paper, since it simulates (in the sequential way) the run of an accelerator (called ω-matrix) described in Section 5.

It should be noted that procedures for operations Difference and Semi-join can be constructed similarly to the procedure INTERSV.

Now we consider the operations Product and Join. They assemble a new relation which should be located in a new area of the memory module. For simplicity, we assume that the resulting relation has no more than 256 rows.

Consider the operation Product. Its result relation is obtained as the concatenation of all combinations of the argument relations T and R.

```
proc PRODUCT($T, R$:table; TWF,RWF:slice; var $P(P1, P2)$:table);
var $i, p, r, s$:integer; $X, Y$:slice; $G$:table;
begin $X$:=TWF; $Y$:=RWF; $p := 0$; $r$:=NUMB($X$); $s$:=NUMB($Y$);
  while SOME($X$) do
    begin $i$:=STEP($X$); $p := p + 1$; WCOPY($T(i), s, 1 + (p - 1) \cdot s, P1$)
    end;
*$s$ copies of any tuple of $T$ are created in $P1$*
  COMPACT($R, Y, G$); TCOPY($G, r, P2$);
*$r$ copies of the matrix $G$ are created in $P2$*
end
```

Explain the procedure PRODUCT. Let r be a cardinality of T, s be a cardinality of R and G be a matrix obtained by compaction of R. The procedure PRODUCT constructs s copies of any tuple of T in $P1$ and r copies of the matrix G in $P2$. Note that each attribute of P consists of $r \cdot s$ tuples.

Let us recall the definition of the operation Join. Assume that the first argument relation A has the attributes $A1$, $A2$ and the second one B has the attributes $B1$, $B2$. Let $A2$ and $B2$ be drawn from the same domain. The operation Join concatenates those tuples from its argument relations for which the corresponding values of $A2$ and $B2$ are equal. Now consider the operation Join.

```
proc JOIN ($A(A1, A2), B(B1, B2)$: table; AWF,BWF: slice ;
var $C(C1, C2)$: table );
var $p, m, n, t$: integer; $w$: word; $M, N, Q$: slice; $E1, E2$: table;
begin $M$:=AWF; $t := 0$; CLEAR($C1$); CLEAR($C2$);
  while SOME($M$) do
    begin $p$:=FND($M$); $w := A2(p)$; MATCH($A2$,AWF,$w, N$); $M := M \oplus N$;
* All occurrences of $w$ are deleted from $M$*
      MATCH($B2$,BWF,$w, Q$); if SOME($Q$) then
        begin PRODUCT($A1, B1, N, Q, E1, E2$); PUSH($E1, t, C1$);
          PUSH($E2, t, C2$); $m$:=NUMB($N$);
            $n$:=NUMB($Q$); $t := t + m \cdot n$
        end;
    end;
end
```

Let w be a tuple value belonging both to the domain of $A2$ and to the domain of $B2$. The occurrence positions of w are stored both in the slice N for the attribute $A2$ and in the slice Q for the attribute $B2$. Then the selected tuples from the relations A and B are concatenated by the Product operation. The obtained matrices $E1$ and $E2$ are stored into the result matrix (relation) C using the auxiliary procedure PUSH.

There are different variants of the Join operation. Here we have considered the variant when the condition for joining attributes $A2$ and $B2$ is their equality. In the examined procedure Join the condition for joining is written by means of MATCH$(B2, BWF, w, Q)$. Thus, replacing the procedure MATCH$(B2, BWF, w, Q)$ by another condition for joining we can write another variant of the Join procedure. Certain procedures for different conditions for joining were considered in [15].

5. Two specialized processors based on cellular structures

It is known that the relational data model is distinguished by inherent parallelism which allows one to gain in efficiency by means of implementing dedicated homogeneous parallel processors. Two such processors are discussed in this section.

5.1. λ- compressor

The λ-structure [16] is a two-dimensional homogeneous array (Figure 1.a) each cell of which contains two logical gates, AND and OR (Figure 1.b) and realizes logical functions $z' = zt$ (the horizontal channel) and $t' = z \vee t$ (the vertical channel).

Let an arbitrary binary vector be applied to the inputs z of the left boundary of λ-matrix.

Consider the first (the left) column of the matrix. The variable t retains its initial value 0 in the vertical channel of this column only till $z = 0$. In some i_1-st cell, where $z = 1$ is encountered for the first time, the value of t changes to 1 which cannot change then till the lower bound. However, the i_1-st cell receives yet the signal $t = 0$. Hence, it is the single cell in the whole column where the combination $z\bar{t} = 1$ is present. This combination may serve as an indication for extracting the "one".

The gorizontal channel of thus indicated i_1-st cell is closed by the signal $t = 0$. Hence, the first "one" of the given vector does not propagate further along the current row. In all cells lower than the indicated one, $t = 1$, so that $z' = z$. Thus, to the inputs of the second column a duplicate of the given vector is applied, except for its first "one".

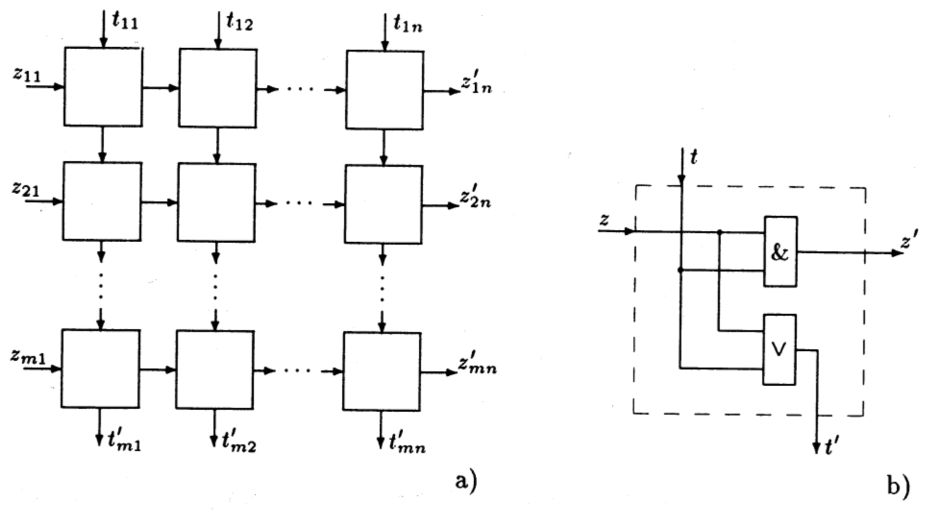

Figure 1. Compression of binary vectors: a) general structure of λ-matrix; b) logical circuit of λ-cell

Similar transformations are performed in the second, the third column, and others: in some i_2-nd cell of the 2-nd column the second "one" of the given vector is indicated, in some i_3-rd cell of the 3-rd column the third "one" is indicated, etc. ($i_1 < i_2 < \ldots$).

Evidently, signals "1" appear at the outputs t' of the lower bound in the 1-st, 2-nd, ... columns of the λ-matrix, and the number of such columns corresponds to the number of ones in the given binary vector.

Hence, the λ-structure performs compression of a binary vector.

Now, introduce into the λ-structure (Figure 1) the second horizontal channel (a bus) f, in each of its rows, and the second vertical channel g realizing the function $g' = g \vee z\bar{t}f$, in each column. So, we obtain the following system of logical functions for each cell:

$$z' = zt, \tag{1}$$
$$t' = z \vee t, \tag{2}$$
$$g' = g \vee z\bar{t}f, \tag{3}$$
$$f' = f, \tag{4}$$

Let us call the resulting scheme λ-compressor. The λ-compressor performs various special functions of interconnection network.

Let boundary inputs of the left boundary be the inputs of the interconnection network and boundary outputs g' of the lower boundary be its outputs. A binary control vector Z is fed bit-wise to the left boundary inputs z, in which the necessary input channels are indicated by ones. The commutation is performed as follows. In each cell, where the combination $z\bar{t} = 1$ is fulfilled, according to (3) we have $g' = f$, i.e., the interconnection function "fork down" is realized. If in the control vector Z the ones occupy the i_1-st, i_2-nd, ..., i_k-th positions, then, as follows from (1)–(2), the combination $z\bar{t} = 1$ appears in the i_1-st row of the 1-st column,the i_2-nd row of the 2-nd column, ..., the i_k-th row of the k-th column. Thus, the outputs g'_{mj} of the 1-st, 2-nd, ..., k-th columns will be connected with the inputs f_{i1} of the i_1-st, i_2-nd, ..., i_k-th rows, correspondingly.

The λ-compressor can be used as a specialized device to support various known data compression algorithms (see, for instance, [17]). Indeed, if at some step of encoding the source data a binary control vector is produced denoting the substrings which should be deleted, this procedure can be easily implemented in the λ-compressor. Moreover, as shown in [16], the λ-structure realizes also *the extension* (insertion of blanks at the given points of a source string) and *the interlacing* (construction of a new string by interspersing the given substrings of two different source strings). It is obvious that the latter two procedures are also useful in implementing data compression.

In the present paper, however, we consider another specific application of the λ-compressor, namely, the deletion of non-relevant tuples while forming the intermediate or final results of relational algebra operation.

5.2. Set intersection processor (ω-matrix)

Another example of a specialized processor for non-numeric processing is the Set Intersection Processor (SIP). It is a two-dimensional homogeneous structure of size $m_1 \times m_2$ (Figure 2), each cell of which contains an equivalence circuit, a response flip-flop, and some additional logic, which is needed for implementing in this cell the sequential bit-wise comparison of the corresponding elements of argument arrays M_1 and M_2. After completing the comparison cycle of n steps, where n is the element length, in the two-dimensional response field of SIP the resulting Binary Label Matrix (BLM) is formed.

The presence of "1" in the (i, j)-th node of BLM means that the i-th element of the array M_1 coincides with the j-th element of the array M_2.

The SIP is a quasi-associative processor with a higher level of parallelism compared to conventional associative processors. Whereas in the conventional quasi-associative processors all elements of an argument array coinciding with *one comparand* are singled out during one cycle of memory interrogation, in SIP *a complete intersection* of two arrays is realized at the same time.

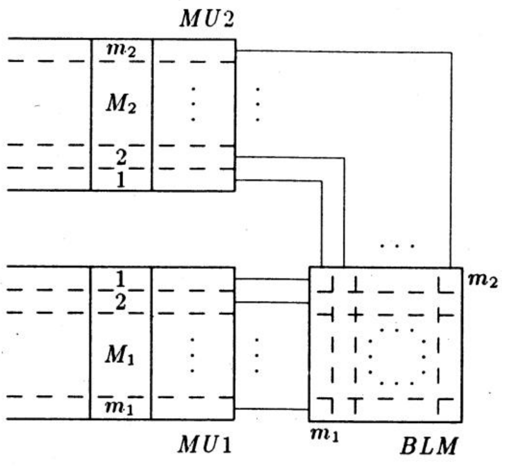

Figure 2. Set intersection processor

6. Representation of relational algebra operations by means of λ-compressor and set intersection processor

In this section we consider the implementation of relational algebra operations from Section 4 using the hardware support from Section 5. To describe the run of the mentioned above ω-processor we extend the language STAR by introducing the following new data type ***syntab***.

Let G be a variable of the type *syntab*. Then G is associated with a matrix G. Its column number and row number are defined by means of the functions Size(G) and Row(G), respectively. The access to the contents of the matrix G will be synchronous both for the rows and the columns.

Note that the type *table* is used for the data array which is stored in the memory blocks, whereas the type *syntab* is used for the ω-matrix.

Introduce a statement of synchronous processing.

Let i be a variable of the type *integer*, A be a variable of the type *array*, G be a variable of the type *syntab* and $S(G,i)$ be a statement. depending on the variable i, but not changing i and A. The synchronous processing statement has the following form:

for all i from A do $S(G,i)$ *od*;

Let us explain the semantics of this statement. The statement $S(G,i)$ is simultaneously performed for all components of A. To describe the run of the set intersection processor we use the following three auxiliary procedures which are realized by hardware.

Consider the procedure INIT*.

```
proc INIT*(Y:slice; var S:syntab);
var i, k:integer;
begin k:=size(S); for all i from [1..k] do
   col(i, S) := Y od
end
```

Explain the procedure INIT*. The contents of the slice Y is synchronously stored into all columns of the matrix S.

Consider the procedure DIS*.

```
proc DIS*(S:syntab; var Z:slice);
var i, r:integer;
begin r:=row(S); for all i from [1..r] do
   Z(i):=dis(S(i)) od
end
```

Explain the procedure DIS*. For all rows $w \in S$ the values dis(w) are synchronously defined.

Consider the procedure LINE*.

```
proc LINE*(A, B, C:slice; var S:syntab);
var i, r:integer;
* Assume that NUMB(C)=size(S) *
begin r:=NUMB(C); for all i from [1..r] do
   if B(i) =' 1' then col(i, S):=col(i, S) ∧ A else
      col(i, S) := col(i, S) ∧ ¬A od
end
```

Explain the procedure LINE*. All columns of S change their contents synchronously. In addition, the new contents of the i-th column of the matrix S depends on the i-th component value of the slice B. Note that the implementation of this procedure is executed by hardware as a unique process of comparison of the i-th columns of two matrices.

Now we consider the procedure INTERS* using the set intersection processor.

```
proc INTERS*(T, R:table; TWF,RWF:slice; var Z:slice);
var S:syntab; X, Y, M:slice; i, k:integer;
begin k:=size(T); size(S):=row(R);
   M:=RWF; INIT*(TWF,S);
   for i:=1 to k do
```

```
        begin $X$:=col($i,T$); $Y$:=col($i,R$);
          LINE*($X,Y,M,S$)
        end;
        DIS*($S,Z$)
  end
```

Explain the procedure INTERS*. At first the matrix S is initialized by means of the contents TWF. Then the procedure LINE* is executed for the i-th column of T and i-th column of R, where $i = 1, 2, \ldots, \text{size}(T)$. After the cycle termination each component $Z(i)$ of the resulting slice Z is obtained as a disjunction of all row components $S(i)$.

It is obvious that using the set intersection processor we can write procedures for the relational algebra operations Difference and Semi-join by analogy with INTERS*.

By means of the λ-compressor the following procedure COMPACT* can be written.

```
proc COMPACT*($T$:table; $Y$:slice; var $H$:table);
var $i, k$:integer; $X, Z$:slice;
begin $k$:=size($T$); size($H$) = $k$;
  for $i$ := 1 to $k$ do
    begin $X$:=col($i,T$); $Z$:=PRESS($X,Y$); col($i,H$) := $Z$
    end;
end
```

It should be noted that the λ-compressor allows one to execute the matrix compaction by means of the vertical processing.

The procedure PRODUCT* is obtained from the procedure PRODUCT replacing the occurrence of the procedure COMPACT by COMPACT*. Similarly the procedure JOIN* is obtained from the procedure JOIN replacing the occurrence of PRODUCT by PRODUCT*.

Note that the resulting relation of the operation Projection is a subset of its argument relation $T(T1, T2)$. But using the λ-compressor we can construct a procedure PROJECT1 (or PROJECT2) having a resulting matrix which is obtained after sewing the identical tuples in the domain of the attribute $T1$ (or $T2$).

7. Conclusion

We compare complexity of algorithms from Sections 5 and 6 which realize the same operations of the relational algebra.

Let us fix two relations T and R. At first we compare complexity of algorithms realizing the procedures PRODUCT and PRODUCT* (or JOIN and JOIN*). It can be easily seen that it is sufficient to compare the complexity of algorithms COMPACT and COMPACT*.

It is not difficult to calculate that $N(\text{COMPACT}) = 3\cdot\text{row}(T)+1$ and $N(\text{COMPACT}^*) = 3\cdot\text{size}(T)$. Note that for real tasks $\text{row}(T) \gg \text{size}(T)$. Therefore

$$N(\text{COMPACT}) \gg N(\text{COMPACT}^*).$$

This result justifies the use of the additional hardware λ-compressor.

Now we compare complexity of algorithms realizing the procedures INTERSV and INTERS* to evaluate the acceleration which is obtained by using the set intersection processor. From the definition of the operation Intersection we have that $\text{size}(T)=\text{size}(R)$. In Section 4 we have obtained that

$$N(\text{INTERSV}) = 1+\text{row}(R) + 2\cdot\text{row}(T) + 3\cdot\text{size}(R)\cdot(1+\text{row}(T)).$$

By the assumption $\text{row}(R) \le \text{row}(T)$. Therefore

$$N(\text{INTERSV}) \le 3\cdot\text{row}(T) + 3\cdot\text{size}(R)\cdot(1+\text{row}(T)).$$

It can be calculated that $N(\text{INTERS}^*) = 3 + 3\cdot\text{size}(T)$. To economize the additional hardware, we execute the procedure COMPACT* for the relation R before the INTERS* run. Therefore we obtain that $N'(\text{INTERS}^*) = 2 + 6\cdot\text{size}(T)$. Hence, $N'(\text{INTERS}^*) \ll N(\text{INTERSV})$ which justifies the use of the set intersection processor.

References

[1] J.L. Potter, W.C. Meilander, Array processor supercomputers, Proceedings of the IEEE, Vol. 77, No. 12, 1989.

[2] W. Haendler, Ya.I. Fet, Vertical processing in parallel computing systems, Proc. of the Intern. Conf. "Parallel Computing Technologies", Novosibirsk, USSR, 1991.

[3] E. Ozkarahan, Database Machines and Database Management, Prentice-Hall, Inc., 1986.

[4] C. Fernstrom, J. Kruzela, B. Svensson, LUCAS associative array processor. Design, programming and application studies, Lecture Notes in Computer Science, Springer-Verlag, Berlin, Vol. 216, 1986.

[5] M.R. Muraszkiewicz, Cellular array architecture for relational database implementation, Future Generations Computer Systems, Vol. 4, No. 1, 1988.

[6] B. Czejdo, R. Elmasri, M. Rusinkiewicz, D.W. Embley, A graphical data manipulation language for an extended entity-relationship model, IEEE Computer, March, 1990.

[7] J.L. Potter, Associative Computing: A Programming Paradigm for Massively Parallel Computers, Plenum Press, New York, 1992.

[8] C.C. Foster, Content Addressable Parallel Processors, Van Nostrand Reinhold Company, New York, 1976.

[9] J. Miklosko, R. Klette, M. Vajtersic, J. Vrto, Fast algorithms and their implementation on specialized parallel computers, Special Topics in Supercomputing, Vol. 5, North–Holand, 1989.

[10] A.D. Falkoff, Algorithms for parallel-search memories, J. of the ACM, Vol. 9, No. 10, 1962.

[11] A.Sh. Nepomniaschaya, Language STAR for associative and parallel computation with vertical data processing, Proc. of the Intern. Conf. "Parallel Computing Technologies", Novosibirsk, USSR, 1991.

[12] K. Jensen, N. Wirth, PASCAL User Manual and Report, Springer–Verlag, Berlin, 1978.

[13] J.D. Ullman, Principles of Database Systems, Computer Science Press, 1980.

[14] A.Sh. Nepomniaschaya, A language STAR for associative and bit-serial parallel processors and its application to relational algebra, Bulletin of Novosibirsk Computing Center, Series: Computer Science, 1, 1993.

[15] A.Sh. Nepomniaschaya, Investigation of associative search algorithms in vertical processing systems, Proc. of the Intern. Conf. "Parallel Computing Technologies", Obninsk, Russia, 1993.

[16] Ya.I. Fet, Parallel Processors in Control Systems, Energoizdat, Moscow, 1981 (in Russian).

[17] J.A. Storer, Data Compression: Methods and Theory, Computer Science Press, Rockville, 1988.