RAP.2—An Associative Processor for Databases and Its Applications

STEWART A. SCHUSTER, H. B. NGUYEN, ESEN A. OZKARAHAN, AND KENNETH C. SMITH, FELLOW, IEEE

Abstract—**RAP—a Relational Associative Processor—is a back-end or peripheral device to augment a general purpose computer for implementing a database management system (DBMS). Its architecture is based on the fact that database operations are inherently set-oriented and that data base addressing is best accomplished through associative reference to achieve high data independence. RAP utilizes these characteristics by combining the features of associative and array processors. Previous publications on RAP have dealt separately with the details of the first version of its architecture [1]–[4] language interface [5], [6] and performance evaluation [7]–[9]. This paper provides details on a recently evolved, faster, and more flexible architecture for RAP called RAP.2 [17].**

Index Terms—**Access methods, array processors, associative memories, associative processors, bubble memory, cellular memories, charge-coupled device (CCD) memory, computer architecture, database machines, database management systems, disk memory, microprocessors, parallel processors, random access memory (RAM), secondary storage devices.**

I. Introduction

RAP—a Relational Associative Processor—is a device designed to act as an attached, peripheral, or back-end machine to augment a conventional computer in providing fast responding and user oriented database management systems. The basic architecture of an RAP device consists of a set of identical components called *cells*, a statistical arithmetic unit, and central controller. This organization is shown in Fig. 1. Each cell is composed of a processor and block addressable memory. The processor is specifically constructed for database definition, insertion, deletion, update, and retrieval primitives. Logic for each processor has been designed to be compatible with large-scale integrated (LSI) circuit implementation technology. The memory can be implemented by a rotating magnetic device such as the track of a disk or drum, semiconductor charge-coupled device (CCD) or random access memory (RAM), or bubble memory. The statistical arithmetic unit is actually part of the controller and is designed for computing summary statistics (e.g., totals, averages, etc.) over the combined contents of the cell memories. The controller is responsible for receiving instructions in RAP machine format from a general purpose front-end or host computer, decoding them, broadcasting control sequences to initiate cell execution, and passing retrieved or inserted items between the front-end, and RAP. Each RAP instruction is executed within the cells which operate in parallel directly on the data. Simple intercell communication for priority polling is implemented along the chain. Each memory contains data formatted into a sequence of records containing values of data items. The details will be given shortly.

A cell is composed of several logic units, the most important being involved with searching. Several comparator elements form the basis of the associative addressing architecture of a cell. The comparators can independently test the contents of one item in the database against several literals or several items each against different literals. The true or false results of comparison tests on a record can be combined into a disjunctive or conjunctive result to determine if the record associatively qualifies for further manipulation.

The key to RAP's performance for database management is parallel processing. Parallel processing eliminates the need for indicies, such as inverted lists or *B*-trees, for fast retrieval. Thus, the maintenance of such access structures are also eliminated. A second advantage to the RAP approach is it provides hardware mechanisms that can interpret memory formats that match the users view of database files and records. This minimizes the software required to translate programming languages into machine instructions. A demonstration is given in the Appendix showing the RAP prototype actual operation.

The front-end computer supports high-level user functions. It interfaces users to RAP by supporting communications via interactive terminals or through programming language CALL and I/O statements for application programs running in multiprogramming operating systems. The translation of various query languages into RAP programs will also be accomplished in the front-end. Database system software responsible for coordinating multiple and diverse secondary storage devices other than RAP, scheduling of queries, and maintaining protection, security, and integrity must also be supported in the front-end but can be aided by the data-processing capabilities of RAP.

Designs for devices similar in philosophy to RAP can also

Manuscript received May 23, 1978; revised November 19, 1978. This work was supported by the Department of Communications, Department of Supply and Services, and the National Research Council of Canada. An earlier version of this paper was entitled "RAP.2—An Associative Processor for Data Bases" and originally presented at the 5th Annual Computer Architecture Symposium, Apr. 3–5, 1978.

S. A. Schuster is with Tandem Computers, Inc., Cupertino, CA 95014.

H. B. Nguyen and K. C. Smith are with the University of Toronto, Toronto, Ont., Canada.

E. A. Ozkarahan is with Middle East Technical University, Ankara, Turkey.

0018-9340/79/0600-0446$00.75

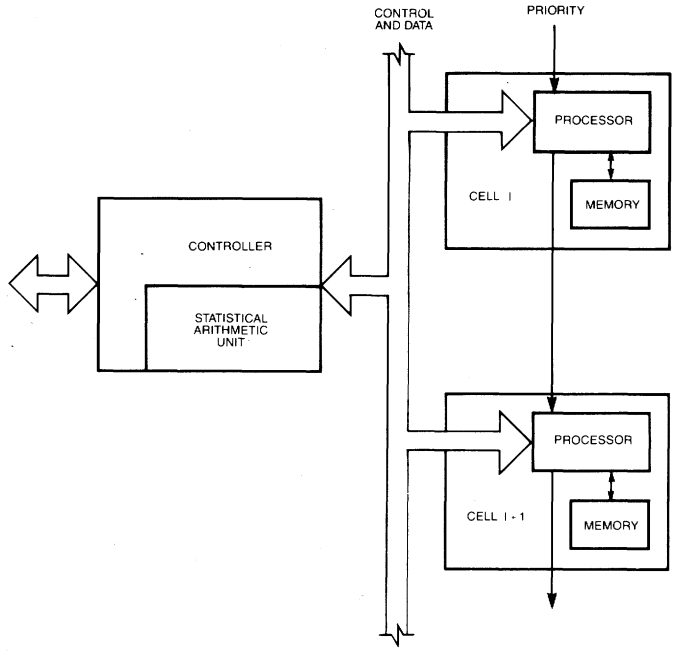


Fig. 1. RAP.2 system architecture.

be found in the literature [10]–[14]. A survey of recent database machine developments is given in [19].

II. The Abstract Machine

The RAP system has a machine-oriented yet high-level and complete assembler instruction set for manipulating databases. Most instructions correspond to one machine instruction which invokes several cell microcode instructions. In this section, an explanation of the RAP assembler instructions will be presented. A programmer's view of the RAP data structure will be given first. Then the basic structure of an RAP instruction will be given followed by the description of each individual instruction.

A. Data Structure

From a programmer's view, RAP stores data as unordered occurrences of records defined by an *RAP relation* as shown in Fig. 2. A relation can be envisioned as a formatted table of data where rows of the table represent a set of record occurrences, sometimes called *tuples* in relational terminology. The occurrences of a relation stores data about a set of similar entities (e.g., persons, places, things, or relationships). The name of a relation identifies the set of entities. The format of record occurrences is defined by naming the *data items* whose concatenated values occur in each record and specifying their length. The length of each item in the relation is fixed according to a user's choice of one of several sizes. Each occurrence of a relation stores data which describes a particular entity by assigning a value to each of the items according to the format of the relation. The values are treated internally as simple bit patterns for nonnumeric data and as integers in two's-complement format for numeric data.

Each relation and its occurrences are augmented by several special one-bit items Mi called *mark bits*. These items can be set to $\emptyset$ or 1 under user control through various marking instructions or by the intermediate operations of other instructions. The bits are used primarily as a work area to temporarily indicate subsets of record occurrences so that the results of one instruction can be used in subsequent instructions. This is done by treating the mark bits as normal data items to be tested during associative addressing. An extra mark bit called the delete flag, which is transparent to users, is provided to indicate deleted tuples to be ignored during instruction execution.

The records of a relation can occupy one or several cell memories, but each cell can only store records from one relation. Therefore, a single RAP device can contain record occurrences from one large relation or from N relations, one for each of N cells. (This is an artificial limitation imposed by our implementation. Allowing records from more than one relation to reside on a cell can increase performance in many cases by having more cells process an instruction.) The programmer of a query need not be aware of the cell location or number of cells occupied by the relations. However, there are occasions, such as during garbage collection or bulk loading, where the user needs to control the device at the cell level. To permit this, a user can refer to registers containing an integer address identifying each cell.

Several registers are also available in the controller. These can be used to store intermediate computations or retrived data from relations and used as search values to be tested in subsequent instructions qualifications for generating complex queries.

An RAP relation is an intermediate-level abstraction of large databases. Although it has a flat tubular structure, it is not quite relational as defined by Codd. For example, duplicate records are permitted and their existence is not

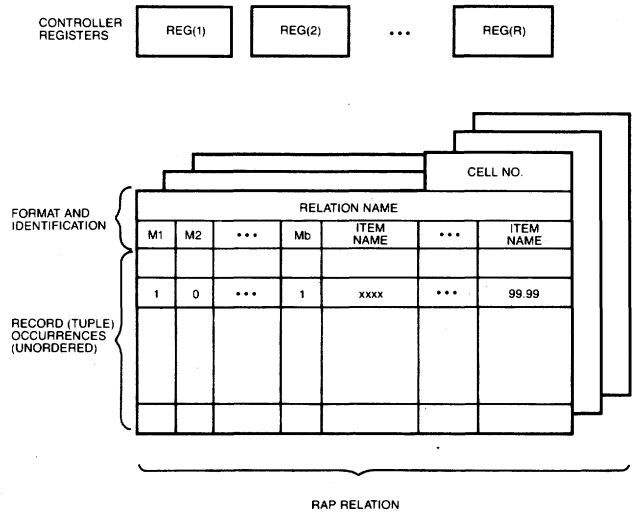


Fig. 2. RAP.2 logical data types.

automatically detected. There are physical limitations on sizes and numbers of items. Also, the special hardware operations for mark bit manipulation is a form of hardwired "access method" that a user must control via program instructions to select desired data for further processing. What RAP does is to provide a data model that is high level but flexible or general enough to easily support software implementations of set-oriented versions of common models such as *hierarchies*, *networks*, and *relations*.

B. Instruction Format

The general format of most RAP instructions is

⟨label⟩⟨opcode⟩⟨mark option⟩

[⟨object⟩:⟨qualification⟩][⟨parameter⟩].

Exceptions will be noted as they arrive. The label is an optional symbolic instruction address. The opcode specifies the data manipulation operation. A mark option can take one of the following forms:

1) ⟨null⟩ implies no marking is done.

2) MARK (⟨bit specification⟩) sets (to "1") the mark bit data items specified in the bit specification of the qualified tuples.

3) RESET (⟨bit specification⟩) resets (to "∅") the mark bit data items specified by the bit specification of the qualified tuples.

The individual mark bits will be denoted M1, M2, ⋯, Mb where b is a hardware parameter limiting the number of mark bits. A bit specification is simply a list of mark bit names. An object has one of the following formats and is used primarily to specify which cells, relations, and individual items are to be manipulated by the instruction:

1) Rn (D1, ⋯, Ds) where Rn is a relation name and (D1, D2, ⋯, Ds) is a list of data item names associated with relation Rn. The data item list is optional or not relevant in many instructions. The index s has a hardware limit on the number of domain names that can be included for certain instructions.

2) List of cell addresses, CELL (i), where i the integer address of the i-th cell.

A qualification in the RAP instruction format can take one of the following forms:

1) ⟨null⟩ implying every tuple of the relation qualifies.

2) Q1 & Q2 & Q3 ⋯ & Qp denoting the conjunction of simple condition Qi.

3) Q1 | Q2 | Q3 ⋯ | Qp denoting the disjunction of simple conditions Qi.

A simple condition Qi can be any one of the following:

1) ⟨Di⟩⟨comparator⟩⟨operand⟩ where
   a) Di is a data item name
   b) comparator is one of $:=$, $\neq$, $<$, $\leq$, $>$, $\geq$
   c) operand is one of REG (i), ⟨integer⟩, "⟨literal⟩," where REG (i) refers to the contents of the i-th controller register.

2) MKED (Mi) denoting the mark bit test Mi = 1.

3) UNMKED (Mi) denoting the mark bit test Mi = 0.

4) CELL (i) indicating that the cell address is tested as part of the qualification.

A qualification has certain restrictions which are imposed by a particular hardware implementation. A qualification can have at most k simple conditions of type 1) (i.e., data item comparisons) and b simple conditions of types 2) and 3) together. Only one simple condition of type 4) may be included in any qualification.

The format of parameter varies greatly and will be explained along with each instruction that requires additional information not supplied above.

C. Description of RAP Instructions

The following is a description of each opcode provided by RAP and an indication of its execution time. Execution times depend on the speed of the cell processor, the capacity of cell memory, and could also vary greatly depending on the choice of technology or architecture of the processor and memory. However, we can give a summary in terms of the number of searches or scans of cell memory required to execute an instruction. The syntax of each instruction is

given followed by the number of memory scans in parentheses required for execution.

1) Selection:

Select⟨mark option⟩[Rn:⟨qualification⟩] (1)

This instruction selects qualified tuples from the relations Rn and sets or resets the mark bits of these tuples according to the mark option given. For example, the instruction:

Select Mark(M1M2) [R1:D1 = "a"]

will set mark bits M1 and M2 of tuples in R1 which have D1 = "a".

Cross-Select⟨mark option on R1⟩[⟨R1⟩:⟨D1⟩⟨comparator⟩⟨R2⟩.⟨D2⟩]
[⟨R2⟩⟨mark option on R2⟩:⟨qualification⟩]
(1 + # of source tuples/k)

This instruction involves operation between two relations called source (R2) and target (R1). It works like a repetitive select instruction on the target relation with the exception that qualification for each selection is obtained from the source relation data item values. That is, in order to select a target relation (R1) tuple, the items D1 and D2, respectively, of target and source relation must have comparable values (i.e., values of the same data format) that satisfy at least one of the comparisons between them. The source tuples participating in the comparison are those which satisfy the second qualification.

2) Retrieval:

Read-All⟨mark option⟩

[Rn(D1, ···, Ds):⟨qualification⟩][⟨work area⟩] (1)

This instruction transfers data from all tuples of Rn satisfying the qualification to the supporting processor's storage address as specified by work area. This could be a sequence of primary memory addresses or a file designation. If the object data item list is present, only those item values are read out, otherwise, the entire eligible tuple is transferred. If the mark option is present, the mark bit items of the eligible tuples will be set or reset according to the given mark option.

Read(n)⟨mark option⟩

[Rn(D1, ···, Ds):⟨qualification⟩][⟨work area⟩] (2)

This instruction is very similar to the Read-All instruction, except that only data items from the "first" n or less qualified tuples are transferred to the supporting processor's storage location. The mark option will only be exercised on the tuples that are transferred.

Save(n)⟨mark option⟩[Rn(D1, ···, Ds):⟨qualification⟩]
[⟨register list⟩]
(2)

Save transfers data items from qualified tuples of a relation to registers of the RAP controller. Only items from the "first" n or less eligible tuples are transferred. If the mark option is present, the mark bits of the tuples will be set or reset according to the mark option. If the data element list is not present, the entire tuple will be transferred, otherwise, only those items in the list are read into the registers. Values will be stored left justified and padded on the right with blanks. Data elements with arithmetic domains will be assumed to be a fixed word length in two's-complement format. This register list can take on any combination of the following two forms:

1) Reg (i), Reg (j), ···, Reg (k)
2) Reg (i) – Reg (j)

where Reg (i) – Reg (j) means Reg (i), Reg (i + 1), ···, Reg (j). The transfer is done in the order given, that is, the first item in the object list is read into the first register designated in the register list, second item into the second register, etc. The s items are read from each tuple. The first item of the second eligible tuple will be read into the s + 1 register in the register list.

Read-Reg[⟨register list⟩[⟨work area⟩] (∅)

This instruction transfers content of the specified RAP registers to the supporting processor. Register list has the same format as the register list in the Save instruction.

3) Statistical Computations:

⟨sopr⟩⟨mark option⟩

[Rn(Dn):⟨qualification⟩][Reg (i)] (1)

where sopr is one of the statistical function operators *Sum*, *Count*, *Max* or *Min*. The opcode Count counts eligible tuples in the relation Rn and places the result in the register specified. (Dn) is omitted for this statistical function. The other instructions compute the specified function over the numeric domain of item Dn from qualified tuples.

4) Update:

⟨opr⟩⟨mark option⟩[Rn(Dn):⟨qualification⟩][⟨opd⟩]
(1)

where opr is one of the operators *Add*, *Sub* or *Replace* and opd is either a constant, a data item name, or an RAP register. Item Dn in every eligible tuple is operated on by opr and value of opd.

Delete[Rn:⟨qualification⟩] (1)

Tuples of relation Rn qualifying for deletion have their delete flag bit set causing the tuple to be ignored in subsequent operations.

Colgrbg[⟨relation list⟩ and/or ⟨cell list⟩] (1)

This instruction initiates the physical deletion of all delete-flagged tuples of the listed relations and/or listed cells. The data are packed towards the beginning of cell memory leaving garbage accumulated towards the end. The cell list has the same format as a register list. The relation list has the following format:

(R1, R2, ···, Rn)

Space-Count[Rn:⟨cell list⟩][Reg (i)] (1)

This instruction will examine the cells of relation Rn and return a value indicating the number of available spaces in these cells. This value is stored into the given register. Available spaces include both empty tuples and the delete-flagged tuples. If the optional cell list is present, only those cells in the cell list will be examined. All cells in the cell list must belong to relation Rn. This instruction is usually used to test for space before an Insert instruction is used.

Insert (n) [Rn:⟨cell list⟩][⟨work area⟩] (1)

Work area is the front-end processor's program storage location containing the n tuples to be inserted. If the optional cell list is given, the n tuples will be inserted in those cells only. There is an arbitrary hardware upper limit on the number of characters that can be inserted in one Insert instruction which places a limit on n.

5) *Data Definition:*

Destroy[Rn:⟨cell list⟩] (∅)

This instruction deletes the tuples, format, and names from the specified cells of a relation. If a cell list is not present, the relation is removed from all the cells it occupies. A special null relation name is reserved for all blank cells.

Create[Rn:⟨cell list⟩][⟨format⟩] (1)

One execution of this instruction formats each cell in the cell list for relation Rn. Empty tuples are delete flagged on the created cells. Format contains parametric data about the length of the data items stored in a relation.

6) *Register Manipulation:* Only registers containing valid integer values will result in meaningful numeric computations. All register arithmetic will assume a specified word length for operands starting at the leftmost bits of controller registers.

Insert Reg[⟨register list⟩][⟨constant list⟩] (∅)

This instruction will insert the constants into the specified registers. If only one constant is present, this constant will be inserted in all registers of the register list. Otherwise the number of constants must match the number of registers.

Dec-Reg[Reg (i)] or Inc-Reg[Reg (i)] (0)

The instruction Dec-Reg subtracts 1 from the contents of Reg (i) and Inc-Reg adds 1 to the contents of Reg (i).

⟨ropr⟩[Reg (i) [⟨ropd⟩] (∅)

where ropr is one of the operators: *Radd*, *Rsub*, *Rmul*, or *Rdiv* and ropd can either be an integer or another register.

7) *Decision and Transfer:*

BC⟨label⟩, ⟨boolean expression of conditions⟩ (∅)

where BC is the abbreviation for "branch on condition." Condition can be one of the following:

a) ⟨null⟩, this implies the instruction is treated as an unconditional branch.
b) Reg (i) ⟨comparator⟩ Reg (j).
c) Reg (i) ⟨comparator⟩⟨constant⟩.
d) Test (Rn:⟨mark qualification⟩).

If boolean condition is true, branching will take place, otherwise control is given to the next instruction. Condition type d) tests each individual mark bit specified by the mark qualification separately and if the test is met for at least one tuple (not necessarily the same one) of relation Rn then the test is true, otherwise, the test is false. Mark qualification can be either disjunctive or conjunctive.

EOQ

This indicates the end of an RAP program or query.

III. Implementation

A. History

The RAP project began in 1975 in the Computer Systems Research Group at the University of Toronto, and in 1976 produced a prototype system (hereafter called RAP.1) consisting of two cells [4]. The RAP.1 system consisted of a partially hardwired controller and each cell had its own memory *track* where the format and timing of a track was modeled on disk technology. In RAP.1 all components of the system were required to be synchronized by a single clock, all tracks had to be of equal length, and several instructions needing intercell communication required RAP to provide data flow capabilities between all cells. Every operation concerning data on the track took one or several full revolutions.

During the project three important decisions were made to change the organization of RAP.1 which resulted in the design and implementation of RAP.2. First, the controller was to be implemented by a mini/micro computer. Second, the data track was designed around the capabilities of emerging block addressable memories instead of a disk. Third, a more uniform and flexible instruction set with extended marking capabilities was needed. A hardwired implementation of the controller was found to be inflexible and speed was not an issue. The development of a disk system that meets the requirements of an RAP system appears difficult and costly because of synchronization and error correction complexity. Furthermore, it is becoming evident that RAM, CCD, bubbles, and electron beam technologies will eventually cause head per track disks to be phased out.

The use of a general purpose microcomputer as the controller resulted in a major redistribution of the work-load. In RAP.2, the cells were greatly simplified and required to perform only those tasks directly related to their tracks. Because the controller is inherently slow and cannot cope with the speed of the cell, it became important to decouple cell synchronization from the controller. The new work-load distribution also freed every cell from the task of sending data directly to other cells. This can be done through the

controller. Each cell can operate independently of other cells and the controller. As a byproduct, each cell can have its own track length and execute different instructions independent of other cells. RAP.2 now looks like a conventional computer system coordinating the tasks of many cells which are treated as independent peripheral devices attached to a bus.

In the summer of 1977, an RAP.2 prototype of 2 cells and query language software were demonstrated. Each cell contained a million bit CCD track built from Intel's 16K bit 2416 component. The controller was a PDP11/10 and RAP.2 was interfaced to a PDP11/45 as a DMA device via the controller. To make the transition from RAP.1 to RAP.2 as fast as possible, it was decided that only essential changes were allowed. Consequently, the RAP.2 implementation is far from perfect and its performance can be greatly improved. In this paper we refer to enhancements not yet implemented as features of a future RAP.3 system.

B. Physical Data Organization

In review, data are organized into files called *RAP relations*. A relation is a collection of *records* sometimes called *tuples*. Each record is a string of many concatenated fields called *data items* in some fixed order. The number of fiels per record of a given relation is a constant. Every relation and each of its fields have a name stored in a compactly coded form. In RAP, the length of each item must be constant. In the RAP.1 and RAP.2 prototypes, each record was limited to 255 items whose length could only be 1, 2, or 4 bytes of encoded data. In RAP.3, this length can be anywhere from 1 to n bytes (where n should be 32 bytes or greater). Presently, each cell stores data from one relation. If a relation is large they can be allocated to several cells. In RAP.3 this restriction would be removed to allow pages of tuples from several different relations to reside in the same cell. This would allow relations to be spread across more cells maximizing cell parallelism. An analysis of how this effects RAP performance is discussed in [15].

In RAP.1, a cell stored the relation name as well as the cell address at the track head followed by tuples separated by gaps as shown in Fig. 3. A two-bit code was attached to each item in a record to specify its length. In RAP.2, the cell address is defined by an 8-contact switch set by an operator. The relation name is stored in a 16-bit register and is defined by the programmer. Both the cell address and the relation name can be read out. The new format for each tuple remains unchanged except that the two-bit space between 2 consecutive domains is left blank since all the length codes are stored in a register called the length code RAM. As for gaps, the only requirement is that each tuple must fit in an arbitrary integral number of minor loops.

The CCD memories of each cell behave like a very long drum with many small tracks of 256 bits each. In the remaining part of this paper, by "track" we mean the entire CCD drum and each 256-bit circumference will be called a minor loop. RAP.2 simulates a disk read head by the use of a counter which points to the "current" location. The write head position can be calculated from the read head by using an adder. For most instructions, the write head is one data block (a tuple or record plus gaps) behind the read head. Because of the randomly accessible nature of minor loops, access time is small (the worst case is 256 bit times). When a cell idles, its' read head is positioned on the first minor loop. In operation, each instruction requires the heads to scan just enough data to complete the job. After an instruction is completed, the heads immediately return to the first minor loop. Due to this property, it is more appropriate to use the term "scan" instead of "revolution" to indicate the time required to do an instruction. In data retrieval or insertion, a scan ends immediately when a sufficient number of records have been retrieved or inserted.

In RAP.3, storage efficiency would be maximized. There would be no inter-item spaces or gaps. Also, a "return from halt" option (analogous to the "return from subroutine" instruction of some microprocessors) would allow the cell to resume scanning at some previous spot. This option greatly improves execution time where a very large volume of data is to be inserted or retrieved.

C. Global Architecture

The RAP.2 system is organized as shown in Fig. 1. There are 8 control lines and 16 data lines. A DMA link is established between the data bus of the controller and front-end computer. There is a priority line that runs through all cells to allow fast polling of individual cells. This is used to sequence controller access to cells to retrieve accumulated statistical computations without having to poll every cell in the system.

The reason why direct data communications between cells was dropped in RAP.2 is multifold. First, expensive drivers were needed because each cell was required to drive all others. Second, there was the classical transmission line problem requiring the entire RAP.1 system to be crammed into a physically small space. Third, we wanted to desynchronize the system to maximize reliability and concurrency. Last, reliability suffers when data are sent automatically from any cell directly to all others. If one cell is malfunctioning, the whole system could crash and diagnosis would become an extremely difficult task. Furthermore, the amount of information to be exchanged is usually too small to justify the cost of direct communication links.

Out of eight control lines, seven are used to encode a maximum of 128 micro-code commands called "keys." The eighth is called "key enable" and is used to indicate valid data. In practice, these lines are connected to the least significant part of the address bus of the PDP 11 controller and the key enable line is decoded from the most significant part. Some keys are accompanied by an operand which must appear on the data bus, some expect data from cells to be put on the data bus, and others are not associated with any data.

Commands are broadcast to all cells of the system. Establishing a scheme to selectively restrict the execution of

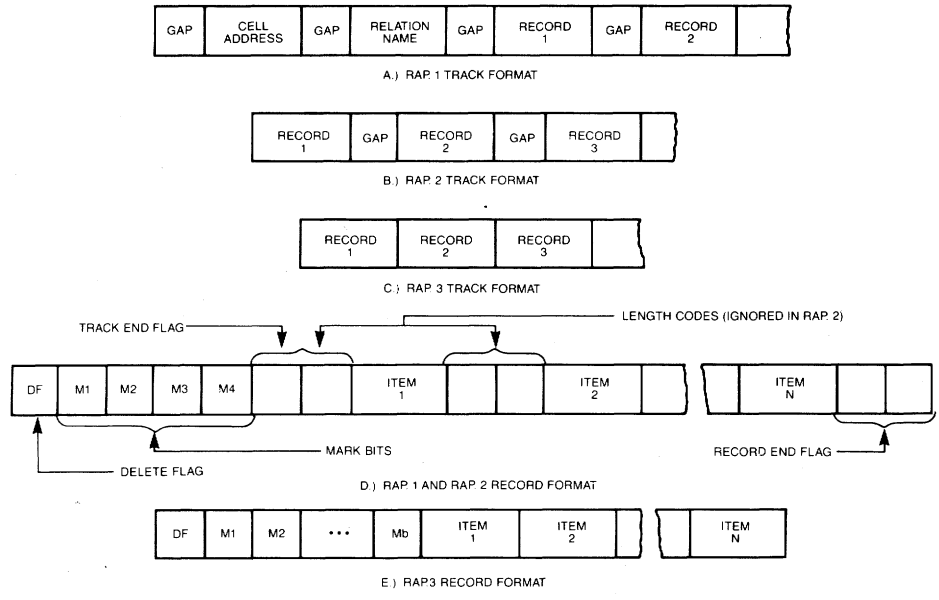


Fig. 3. RAP physical data formats.

commands by a subset of cells is handled by the creation of three state-variables called "open," "blocked," and "rejected." There are three different ways to open a cell. In the simplest case, the controller can open any particular cell by its integer designation by storing that value in one key location. Finally, cells can be opened by referencing the name of the relation stored in the cell. The states "blocked" and "rejected" together with the priority line and the "get next cell" command are used for controlling the opening and closing of cells in a sequence. Consider the following analogy. A number of persons (cells) form a line to buy a ticket in a theater. Those who have bought one are "rejected," those who still have to wait are "blocked." The one who is buy is neither blocked nor rejected. Each time the line move corresponds to a "get next cell" execution. This command rejects the current non-blocked cell and unblocks the first blocked cell. For example, to serve all cells of a relation Rn sequentially, the controller must first open all Rn cells and then blocks them which is achieved by referring to a key. It then sequences through a program loop starting with a "get next cell" followed by the service routine.

For a cell to respond to a command, the cell must be in a proper state; it must be open, neither blocked nor rejected, and furthermore, not running. The last condition is a measure of protection against any erroneous attempt to change the parameters of a query for the nature of the instruction being executed.

For I/O each cell has an (1K-word) RAM called the I/O buffer and a pointer which is resettable by the controller. As far as the controller is concerned, the I/O buffer looks like a single reserved memory location. Every time a word is stored in this location, it is sent to the I/O buffer where a pointer is incremented automatically. To insert a set of tuples, all the controller has to do is repeatedly store 2 bytes at a time in the reserved location. The number of tuples to be inserted is written in another reserved location. After the cell is initiated to run, it will look for vacant slots on its track and pull data from its I/O buffer to fill them.

During data retrieval, the opposite is done. The cell looks for desired data on its track and puts them in its I/O buffer. Since the buffer size is limited, the controller must also indicate how many tuples to be retrieved. For reading, the I/O buffer also looks like a reserved memory location. The buffer pointer is automatically incremented every time this location is read out.

Besides track data, many other kinds of information of a cell are also available for retrieval by the controller: processing status, cell address, relation name, buffer pointer, result register for statistical computations, and the S-counter which contains the number of satisfied tuples in the most recent pass. Access to the buffer pointer allows the establishment of a future DMA link for rapid bulk transfer of inserted/retrieved data directly between cells and the front-end computer.

D. Cell Structure

The structure of a cell can be divided into eight units.

1) Cell Interface: This unit implements the interface to the control and data buses. It contains bus receivers and a large decoder that decodes the contents of the control bus. The cell address is part of this unit and is defined by an 8-contact switch. Also, there is a 16-bit relation name register. The logic for "get next cell" is also part of this unit. As mentioned before, every reference to the related controller key will affect the states "blocked" and "rejected" of an open cell. Finally there are also status states indicating whether in the last pass, bit DF, M1, M2, etc., are marked and a state indicating if there was a satisfied tuple. The most significant bit of the status is always a "1" and is used to indicate the presence of a cell.

2) Synchronizer: This is the largest logic unit of a cell. It provides all timing signals and shift clocks to the rest of the

cell. For simplicity and ease of testing, all basic clock signals are periodic. This implies that there is always a read phase and a write phase for the CCD memories. Consequently, the bit rate of RAP.2 is slightly below 1 MHz. (In RAP.3, read and write phases would be allowed only when necessary.) The 1 MHz bit rate is a limitation of the CCD's and not of the cell logic which has been rated at 10 MHz.

3) *Query Analyzer:* This is the heart of a cell and it determines whether a tuple satisfies a search qualification. The query analyzer has two parts: the terms evaluator and k identical data item comparator units (k = 3 in the prototype). Each comparator unit has an 8-bit register to store the item number to be tested, a tapped 32-bit shift register for the externally supplied constant, a 4-bit register which indicates the selection of the unit and the symbols (<, =, >) of the comparison, and a serial comparator.

4) *I/O Buffer:* This unit is of prime importance to decouple a cell from the controller for data retrieval and insertion. It consists of a 1K × 16 RAM buffer and a collection of pointers.

5) *Arithmetic Unit:* This is the only unit that is not vital to the operation of the rest of the cell. It is only necessary for supporting arithmetic instructions (namely Add, Sub, Sum, Max, Min) and can be removed if they are not required. It contains three tapped shift registers to store operands and results.

6) *Update Control:* This is the smallest unit of a cell. It has a register to store information concerning the Mark and Reset option. It takes care of the marking and resetting of mark bits as well as the writing of new data supplied by the I/O buffer for Insert or by the arithmetic unit for Add, Sub, and Replace. It also erases the track for Create or selected tuples for Delete.

7) *Output Multiplexer:* This is logically the simplest unit. Appropriate registers are connected to various bus drivers which are enabled by signals from the cell interface decoded from the control bus. For most registers, it is the duty of the controller to assure that only one cell at a time is in the readable state otherwise information on the data bus is meaningless. The only exception is the reading of cell status which is meaningful in an "OR" form.

8) *CCD Memories:* Each cell in the prototype contains 1 Mbit of CCD memory. Due to physical limitations, the 1 Mbit drum occupies three identical boards. Each board contains all necessary drivers and 20 Intel 2416 CCD chips which are arranged in an *X-Y* matrix of 4 × 5. Two different kinds of drivers are used: Intel's 5244 chips are used for shift inputs and Intel's 3245 chips for addressing. Currently, all the CCD chips are driven at a same frequency. If the memory size is to be expanded much larger, it makes sense to use two different rates where one of two chips at a time are driven at a fast rate and the rest at the minimum frequency to conserve power.

E. Some Statistics

Each cell breadboard (including the 1 Mbit track) requires about 9 A at 5 V. There is a total of 13 boards employing 412 IC packages (218 SSI + 117 MSI + 77 LSI). For each additional 1 Mbit extension, 96 IC's are needed.

It is clear to us now that we should have implemented the memory system and cell logic to operate word parallel. We would have increased performance greatly and in many cases reduced logic design complexity by being able to use existing components. The bit serial approach was taken because of our initial choice to model head per track disk technology. An example of such an architecture for RAP is presented in [20].

IV. Using RAP in a Database Computer

One might conceive of the day in which microprocessor logic and memory becomes so inexpensive that all secondary memories would have RAP-like processing capabilities. However, the first generation of commercial RAP's would have capacity limitations due to cost relative to the total database storage requirements. A cost effective system would, therefore, consist of a triad of component types: a front-end general purpose computer to interface with users and provide operating system and language processing functions,one or more RAP devices used to act as a file "cache," and one or more conventional secondary memories. This architecture is shown in Fig. 4. With appropriate software, the triad could then be considered to be a specialized *database computer* [13]. We will briefly outline two approaches to the DBMS software organization that exploit such an architecture.

A. Database Partitioning

This approach attempts to exploit the notion that not all data in a data base, at a particular point in time, requires the same processing capabilities. Data can be categorized by the system according to its usage characteristics and placed on the conventional secondary memories or RAP depending on processing requirements that best fir the data. We can partition the database files both horizontally, placing certain records on RAP and others on disk, and/or vertically, placing clusters of data items on one device or the other. Extra data items such as record id's may be required to link corresponding partitions.

The implementation of such a system should include mechanisms for both user controlled and automatic migration of data between the various devices as usage of the data changes. Research into algorithms that exploit database device partitioning is under way at the University of Toronto [16].

A request would be processed by decomposing it into RAP and disk subqueries and first executing the RAP subqueries. Access would then be made to disk only if the request cannot be entirely serviced by RAP. In this case the response from the RAP subquery would be used to minimize the search over the disk portion.

B. Paging and Virtual Memory

This approach exploits the techniques of paging operating systems to provide a virtual associative address space for an RAP device. This requires all the data in the database to be stored according to RAP memory format. The data are then

divided into pages of the size of one RAP cell. All database queries are translated into RAP processing statements. Before execution, each query is directed through a software monitor executing on the front-end computer. The principal tasks of the monitor are to maintain a table that gives the location of the pages for each database file, analyze which pages are required to execute a query, and then page the necessary data between the conventional secondary storage devices and the RAP processor. The query is then passed to the RAP processor for execution. It would be optimum to have a direct path between the secondary storage devices and RAP so that pages would not have to be transferred through front-end.

As opposed to the partitioned approach, all queries will be executed entirely within the RAP processor. A detailed design of the proposed monitor has been outlined in a previous study [9]. For simplification, we require that all the data for a query be small enough to store on the RAP device before being processed. Many of the architectural extensions proposed for the RAP.1 device to allow the overlapping of paging with processing are not required by the RAP.2 architecture.

A GPSS simulation of the entire system was performed and the results were analyzed [3], [7]. Statistics were collected on the average response time for on-line queries for population of Poisson arrivals with a fixed mean and specific sized RAP device. Response was studied with respect to average exponential processing times, average amount of data stored in a relation, total database size, and uniform and exponential *locality* of relation references. Locality was defined as the degree to which short sequences of queries reference some relations more than others. It was found that no significant losses in performance will occur in user environments which exhibit some relative combination of the following characteristics:

1) Relations that occupy a small number of cells.
2) Query populations which exhibit long processing times relative to their paging requirements so that overlapping of processing and paging can be effective.
3) Query populations which exhibit a "significant" amount of locality.

V. Summary of Performance

An important feature of the RAP instruction set is that it is *relationally complete* meaning that any query expressable by the relational calculus can be implemented entirely within the RAP processor [6]. This eliminates the need to transfer extensive amounts of data derived from the intermediate results of query processing between RAP and the supporting computer.

It is important to note that each high-level instruction operates on at most two entire relations during its execution. The hardware is naturally locked during an instruction execution. Thus all software schemes concerned with mutual exclusion of update operations can implement synchronization mechanisms at the relation level. This eliminates much of the operating system overhead incurred by conventional implementations as well as reducing the complexity of maintaining high levels of consistency.

Studies have been conducted to compare the hypothetical performances of the original RAP architecture relative to a conventional computer system for implementing a relational database [8]. Both approaches were modeled analytically. The models considered resident databases for the original RAP architecture and fast access paths in the form of inverted lists for the conventional system. The results show that gains from one to three orders of magnitude in query execution speed can be achieved by the RAP architecture over conventional systems. Furthermore, the new architecture improves this gain substantially through the use of retrieval mechanisms exploiting block addressable memories. The model studied queries of the form: retrievals and updates on records of relations selected with respect to simple and complex Boolean qualifications, retrievals that include statistical criteria in the selection qualification, and retrievals involving the implicit join of two or more relations. This study indicates that, under many circumstances, on-line retrievals and updates of large databases may only be possible with the use of RAP-like systems.

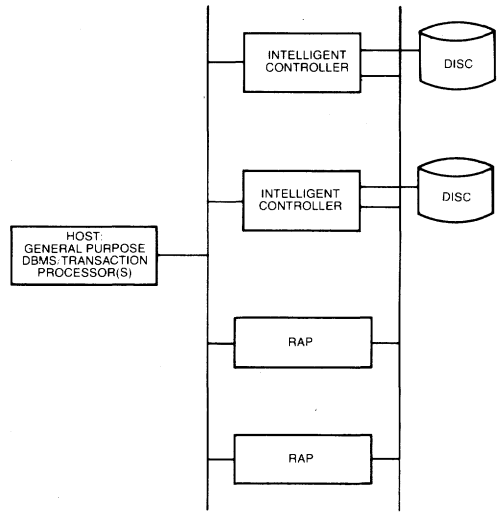


Fig. 4. Using RAP in a database computer.

Appendix

The following are the results of a live demonstration of the RAP.2 prototype hardware. The database contains two relations describing bus drivers and routes. The DRIVER relation has data items DRIVER_NO, SURNAME, INITIAL, HOME town, SEX, MARITAL_STATUS, HEIGHT, BIRTH_YEAR, DAY_HIRED, MONTH_HIRED, YEAR_HIRED, and SALARY. The TRIP relation contains data items TRIP_NO, ORIGIN, DESTN, LV time, AR time, TRAVEL_TIME, MILEAGE, FARE, and DRIVER_NO.

The demonstration includes a monitor running on a PDP11/45, that translates the experimental IBM relational query language SEQUEL-2 into RAP.2 assembler statements [18]. An assembler then generates internal RAP.2 machine code which is transmitted to the RAP.2 peripheral for execution. The host machine is interrupted to accept results and the monitor displays them at the users station. Each query is described in an English comment before being entered as a SEQUEL-2 statement. The RAP.2 assembler statement translation is presented. Following this, execution statistics are displayed. Finally, any tuples or records to be retrieved are displayed.

```
—>FILE 'QUERY3.S'
—>
%    QUERY 3

     CALCULATE THE AVERAGE AND TOTAL MILEAGE DRIVEN BY DRIVERS WHO LIVE
     IN TORONTO, AND RETRIEVE THE TRIP NUMBERS, ORIGINS, AND DESTINATIONS
     OF THEIR TRIPS.

     +SELECT TRIP_NO, ORIGIN, DESTN, AVG(MILEAGE), SUM(MILEAGE)
       FROM   TRIP
       WHERE DRIVER_NO IS IN +SELECT DRIVER_NO
                             FROM DRIVER
                             WHERE HOME = 'TORO'++

THE SEQUEL STATEMENT HAS BEEN TRANSLATED IN 10 RAP INSTRUCTIONS

—>RAP

 1)    SELECT MARK(M1) [DRIVER:HOME = 'TORO']
 2)    CROSS_SELECT MARK(M1) [TRIP:DRIVER_NO = DRIVER, DRIVER_NO] [DRIVER RESET(M1):MKED(M1)]
 3)    SUM [TRIP(MILEAGE):MKED(M1)] [REG(1)]
 4)    COUNT [TRIP:MKED(M1)] [REG(2)]
 5)    RDIV [REG(1)] [REG(2)]
 6)    READ_REG [REG(1)]
 7)    SUM [TRIP(MILEAGE):MKED(M1)] [REG(1)]
 8)    READ_REG [REG(1)]
 9)    READ_ALL RESET(M1) [TRIP(TRIP_NO, ORIGIN, DESTN):MKED(M1)] [QUERY3.0]
10)    EOQ

—>EXECUTE

QUERY TRANSMITTED

QUERY EXECUTION:
        13 REVOLUTIONS TO EXECUTE.
        18 SIXTIETHS OF A SECOND.

AVERAGE(MILEAGE) FROM TRIP IS 147

SUM(MILEAGE) FROM TRIP IS 1615

 11 TUPLES RETRIEVED

—>DISPLAY *

 +TRIP                                          +
 +                                              +
 +TRIP_NO        +ORIGIN        +DESTN          +
 ---------------------------------------------
 +101            +TORO          +LOND           +
 +106            +LOND          +TORO           +
 +201            +HAM           +NF             +
 +206            +NF            +HAM            +
 +300            +TORO          +MONT           +
 +400            +KING          +MONT           +
 +600            +TORO          +OTTA           +
 +601            +OTTA          +TORO           +
 +700            +TORO          +BARR           +
 +705            +NBAY          +BARR           +
 +705            +BARR          +TORO           +

—>FILE 'QUERY4.S'
```

```
—>
%    QUERY 4

     DUE TO INCREASING COSTS, THE FARE OF TRIPS BETWEEN TORONTO AND LONDON
     MUST BE RAISED TO 8
DOLLARS.                                                                    %

     +UPDATE  TRIP
      REPLACE FARE = 8
      WHERE   ORIGIN = 'TORO' AND DESTN = 'LOND'
          OR  ORIGIN = 'LOND' AND DESTN = 'TORO'+

THE SEQUEL STATEMENT HAS BEEN TRANSLATED IN 6 RAP INSTRUCTIONS

—>RAP

 1)      SELECT MARK(M1) [TRIP:DESTN = 'LOND' & ORIGIN = 'TORO']
 2)      SELECT MARK(M2) [TRIP:DESTN = 'TORO' & ORIGIN = 'LOND']
 3)      SELECT MARK(M3) [TRIP:MKED(M2) + MKED(M1)]
 4)      REPLACE RESET(M3) [TRIP(FARE):MKED(M3)] [8]
 5)      SELECT RESET(M1M2) [TRIP]
 6)      EOQ

—>EXECUTE

QUERY TRANSMITTED

QUERY EXECUTION:
        5 REVOLUTIONS TO EXECUTE.
        8 SIXTIETHS OF A SECOND.

0 TUPLES RETRIEVED

—>FILE 'QUERY6.S'

—>
%    QUERY 6

     FOR ALL DRIVERS WHO LIVE IN OTTAWA, GENERATE A LIST OF THE TRIPS
     DRIVEN BY EACH DRIVER, ALONG WITH THE DRIVERS' NAMES AND NUMBERS.      %

     +JOIN ON TRIP.DRIVER_NO = DRIVER.DRIVER_NO
          +SELECT DRIVER_NO, SURNAME
           FROM   DRIVER
           WHERE HOME = 'OTTA'+
     WITH
          +SELECT ORIGIN, DESTN
           FROM   TRIP++

THE SEQUEL STATEMENT HAS BEEN TRANSLATED IN 19 RAP INSTRUCTIONS

—>RAP

 1)      SELECT MARK(M1M2) [DRIVER:HOME = 'OTTA']
 2)      SELECT MARK(M1) [TRIP]
 3)      BC L1, TEST [DRIVER:MKED(M1)]
 4)      SELECT RESET(M1) [TRIP]
 5)      BC END
 6) L1   BC L2, TEST [TRIP:MKED(M1)]
 7)      SELECT RESET(M1M2) [DRIVER]
 8)      BC END
 9) L2   SAVE(1) RESET(M1) [DRIVER(DRIVER_NO):MKED(m1)] [REG(1)]
10)      SELECT MARK(M2) [TRIP:DRIVER_NO = REG(1) & MKED(M1)]
11)      BC L3, TEST [TRIP:MKED(M2)]
12)      BC L4
13) L3   READ [DRIVER(DRIVER_NO, SURNAME):UNMKED(M1) & MKED(M2)] [APPEND QUERY6.0]
14)      RETRIEVE(1) RESET(M2) [TRIP(ORIGIN, DESTN):MKED(M1) & MKED(M2)] [APPEND QUERY6.0]
```

```
15)       BC L3, TEST [TRIP:MKED(M2)]
16) L4    SELECT RESET(M2) [DRIVER:UNMKED(M1) & MKED(M2)]
17)       BC L2, TEST [DRIVER:MKED(M1)]
18)       SELECT RESET(M1) [TRIP]
19) END EOQ

—>EXECUTE

QUERY TRANSMITTED

QUERY EXECUTION:
        63 REVOLUTIONS TO EXECUTE.
        82 SIXTIETHS OF A SECOND.

 7 TUPLES RETRIEVED

—>DISPLAY *

+DRIVER                          +TRIP                    +
+                                +                        +
+DRIVER_NO    +SURNAME    +ORIGIN    +DESTN    +
-------------------------------------------------
+103          +WATT       +TORO      +LOND     +
+103          +WATT       +LOND      +TORO     +
+106          +BEGG       +OTTA      +PETE     +
+106          +BEGG       +PETE      +TORO     +
+118          +BOND       +NBAY      +OTTA     +
+121          +BARR       +WIND      +LOND     +
+133          +CARR       +OTTA      +NBAY     +

—>QUIT
```

ACKNOWLEDGMENT

The authors thank Intel Corporation for their donation of CCD and associated driving components. Also, appreciation is given to Tandem Computers Incorporated and to Mrs. S. Zinker who prepared the final version of this manuscript with the aid of the Tandem 16 text preparation facilities. Most important, the authors thank the members of the RAP project that have contributed so much to its accomplishments: M. Chan, A. Cousin, R. Freen, C. Hawkins, R. Hudyma, H. Huwito, J. Klebanoff, W. Lane, R. Nakano, B. Patkau, P. Pereira, A. Radacz, R. Reid, P. Sadowski, M. Soong, K. Sevcik, and A. Tsonis.

REFERENCES

[1] E. A. Ozkarahan, S. A. Schuster, and K. C. Smith, "A data base processor," Tech. Rep. CSRG-43, Computer Syst. Res. Group, University of Toronto, Sept. 1974.

[2] ——, "RAP—An associative processor for data base management," in *Proc. AFIPS NCC*, vol. 44, 1975, pp. 379–387.

[3] S. A. Schuster, E. A. Ozkarahan, and K. C. Smith, "A virtual memory system for a relational associative processor," in *Proc. AFIPS NCC*, vol. 45, 1976, pp. 855–862.

[4] E. A. Ozkarakan, "An associative processor for relational data bases—RAP," Ph.D. dissertation, University of Toronto, 1976.

[5] L. Kerschberg, E. A. Ozkarahan, and J. E. S. Pacheco, "A synthetic english query language for a relational associative processor," in *Proc. 2nd Int. Conf. Software Engineering*, Oct. 1976.

[6] E. A. Ozkarahan and S. A. Schuster, "A high level machine-oriented assembler language for a data base machine," Tech. Rep. CSRG-74, Computer Syst. Res. Group, University of Toronto, Oct. 1976.

[7] R. Nakano, "A simulator for a RAP virtual memory system," M.S. thesis, University of Toronto, 1976.

[8] E. A. Ozkarahan, S. A. Schuster, and K. C. Sevcik, "Performance evaluation of a relational associative processor," *ACM TODS*, vol. 2, pp. 175–195, June 1977.

[9] E. A. Ozkarahan and K. C. Sevcik, "Analysis of architectural features for enhancing the performance of a data base machine," *ACM TODS*, vol. 2, pp. 297–316, Dec. 1977.

[10] G. P. Copeland, G. J. Lipovski, and S. Y. W. Su, "The architecture of CASSM: A cellular system for nonnumeric processing," in *Proc. 1st Annu. Symp. on Computer Architecture*, 1973.

[11] C. F. DeFiore and P. B. Berra, "A data management system utilizing an associative memory," in *Proc. AFIPS NCC*, vol. 42, 1973.

[12] C. S. Lin, D. C. P. Smith, and J. M. Smith, "The design of a rotating associative array processor for a relational data base management application," *ACM TODS*, vol. 1, pp. 53–65, Mar. 1976.

[13] R. I. Baum and D. K. Hsiao, "Data base computers—A step towards data utilities," *IEEE Trans. Comput.*, vol. C-25, Dec. 1976.

[14] S. G. Zaky, "Microprocessors for non-numeric processing," in *Proc. 3rd Workshop on Computer Architecture for Non-Numeric Processing*, May 1977, pp. 23–30.

[15] P. Sadaowski and S. Schuster, "Exploiting parallelism in a relational associative processor," *Proceedings of the Fourth Workshop on Computer Architecture for Non-Numeric Processing*, Aug. 1978, pp. 99–109.

[16] R. Freen, "A partitioned data base for use with a relational associative processor," M.S. thesis, Department of Computer Science, University of Toronto, Dec. 1977.

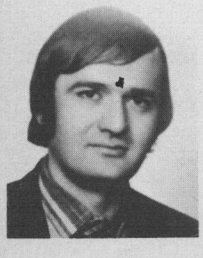
[17] S. A. Schuster, H. B. Nguyen, E. A. Ozkarahan, and K. C. Smith, "RAP.2—An associative processor for data bases," in *Proc. 5th Computer Architecture Symp.*, May 1978.

[18] D. D. Chamberlin *et al.*, "SEQUEL 2: A unified approach to data definition, manipulation and control," *IBM J. Res. Develop.*, vol. 20, pp. 560–575, Nov. 1976.

[19] S. A. Schuster, "Data base machines," *IEEE Proc. Conf. on Computing in the 1980's*, pp. 125–131.

[20] E. A. Ozkarahan and K. Oflazer, "Microprocessor based modular database processors," in *Proc. 4th Int. Conf. Very Large Data Bases*, Sept. 1978, pp. 300–311.

H. B. Nguyen, photograph and biography not available at the time of publication.



Esen A. Ozkarahan received the B.S.E.E. degree from the Middle East Technical University, Ankara, Turkey, in 1962, and the M.S.O.R. degree from New York University, New York, NY, in 1966.

After working in industry at several places in Europe and North America, he received the Ph.D. degree in computer science from the University of Toronto, Toronto, Canada, in 1976. He was involved as a coprincipal investigator of the RAP Project which was related to the Ph.D. dissertation. After working as a Visiting Assistant Professor at the University of Toronto, he returned to the Middle East Technical University and is currently an Assistant Professor of Computer Science. His research interests are database machines and database management systems.

Stewart A. Schuster received the B.S. degree in applied mathematics and computer science from Washington University, St. Louis, MO, and the M.S. and Ph.D. degrees in applied statistics and computer science, respectively, from the University of Illinois at Urbana-Champaign.

He is now a Systems Designer at Tandem Computers Incorporated. Previously, he held positions as a Senior Staff Engineer at Intel Corporation and as an Associate Professor of Computer Science and Business at the University of Toronto. His interests lie in the areas of database management systems, computer architecture, and economics of computers.

Dr. Schuster is a member of the Association for Computer Machinery and the IEEE Technical Group on Database Engineering.

Kenneth C. Smith (S'53–A'54–M'60–SM'76–F'78) was born in Toronto, Ont., Canada, on May 8, 1932. He received the B.A.Sc. degree in engineering physics in 1954, the M.A.Sc. degree in electrical engineering in 1956, and the Ph.D. degree in physics in 1960, all from the University of Toronto, Toronto, Ont., Canada.

From 1954 to 1955 he served with Canadian National Telegraphs as a Transmission Engineer. In 1956 he joined the Computation Centre, University of Toronto, as a Research Engineer assigned to assist in the development of high-speed computers at the Digital Computer Laboratory, University of Illinois, Urbana. In 1960 he joined the staff of the Department of Electrical Engineering at the University of Toronto as an Assistant Professor. In 1961 he returned to the University of Illinois as Assistant Professor of Electrical Engineering where he became Chief Engineer of Illiac II and attained the rank of Associate Professor of Computer Science. In 1965 he returned to Toronto where he is Professor in both the Departments of Computer Science and Electrical Engineering. As Director from 1968 to 1976 and President from 1974 to 1976 of Electrical Engineering Consociates Limited, and presently as a consultant member, he has participated in development programs in a number of Canadian and U.S. industries. He is currently on the board of directors of Matrix Publishers, Champaign, IL, Owl Instruments, Downsview, and Interex Computing Systems Ltd., Toronto. He is currently engaged in the design of digital and linear instrumentation systems.