Database Machines

Fourth International Workshop

Edited by
D.J. DeWitt and H. Boral

Springer-Verlag New York Berlin Heidelberg Tokyo

Database Machines

Fourth International Workshop
Grand Bahama Island, March 1985

Edited by

D.J. DeWitt and H. Boral

With 142 Illustrations and 35 Tables

Springer-Verlag
New York Berlin Heidelberg Tokyo

D.J. DeWitt
Computer Sciences Department
University of Wisconsin
Madison, WI 53706
U.S.A.

H. Boral
Microelectronics and Computer
Technology Corporation
Austin, TX 78759
U.S.A.

(C.R.) Computer Classification: H.2, H.2.6

Library of Congress Cataloging in Publication Data
Main entry under title:
Database machines.
Proceedings of the Fourth International Workshop
on Database Machines.
1. Data base management—Congresses. 2. Electronic
digital computers—Congresses. I. DeWitt, D.J. (David J.)
II. Boral, H. (Haran) III. International Workshop
on Database Machines (4th : 1985 : Grand Bahama Island)
QA76.9.D3D316 1985 005.74'028 85-14792

Printed and bound by Braun-Brumfield, Ann Arbor, Michigan.
Printed in the United States of America.

9 8 7 6 5 4 3 2 1

ISBN 0-387-96200-X Springer-Verlag New York Berlin Heidelberg Tokyo
ISBN 3-540-96200-X Springer-Verlag Berlin Heidelberg New York Tokyo

Database Machines

PREFACE

This volume contains the papers presented at the Fourth International Workshop on Database Machines. The papers cover a wide spectrum of topics including descriptions of database machine implementations, analysis of algorithms and database machine components, architectures for knowledge management, recovery and concurrency control issues, and solutions to the I/O bottleneck problem. As at the previous workshops in Florence, San Diego, and Munich, a diverse collection of countries, universities, research labs, and database machine vendors were represented by the authors and conference attendees. Our thanks go to the authors for writing excellent papers and for their efforts in meeting deadlines, to the VLDB Endowment for its cooperation, and to MCC for all its support. Finally, as usual, it is our secretaries that really deserve the credit for making the workshop a success. We wish to thank Cerise Blair of MCC for taking care of all the arrangements for the workshop and Sheryl Pomraning of the University of Wisconsin for her help in preparing this proceedings.

Haran Boral
David J. DeWitt

March 1985

Table of Contents

Mass Storage Systems

Non-French Filters

Associative Processing in Standard and Deductive Databases

K. Hahne, P. Pilgram, D. Schuett, H. Schweppe, G. Wolf
Corporate Laboratories for Information Technology
Siemens AG, Munich, West Germany

Abstract

Progress in computer technology, microprocessors and storage chip design in particular, has had a major impact on computer architecture. Until now, research in database machine architecture focussed on search accelerators attached to slow background storage devices, and on multiprocessor configurations exploiting the parallelism inherent in database tasks.

The use of a quasi-associative device of large capacity, the Hybrid Associative Store (HAS), in standard and non-standard database applications including the inference subsystem of a deductive database management system is discussed. The HAS has been under development at the research laboratories of Siemens AG since 1983.

1. Introduction

Progress in computer technology, microprocessors and storage chip design in particular, has had a major impact on computer architecture. Until now, research in database machine architecture focussed on search accelerators attached to slow background storage devices, and on multiprocessor configurations exploiting the parallelism inherent in database tasks.

System performance has remained an important issue even in new applications like text, image and speech processing, and in the enhancement of databases by deduction facilities. Technological progress, VLSI in particular, opens up new solutions in all these areas.

"Logic on the chip" is one of the most promising directions: processing power can be integrated into memory chips, thus permitting high-performance implementations of parallel and of associative algorithms. In spite of their associative characteristics, achieved by tightly coupling (simple) processors and memory, such devices are obviously different from, though much more cost effective than, proper associative memory. We will call the former device quasi-associative.

In this paper we are going to discuss the use of devices supporting quasi-associative processing in standard and non-standard DB applications.

We base our discussions on the Hybrid Associative Store (HAS), a quasi-associative device still built with off-memory processors. It has been under development at the research laboratories of Siemens AG since 1983 [Wolf85].

Finally, we discuss the application of HAS in the inference subsystem of a deductive database management system.

2. The Hybrid Associative Store (HAS), Principle of Operation.

This chapter will focus on the HAS philosophy, mentioning hardware details only in passing. - Fig. 2.1 shows the general layout of a computer system with HAS.

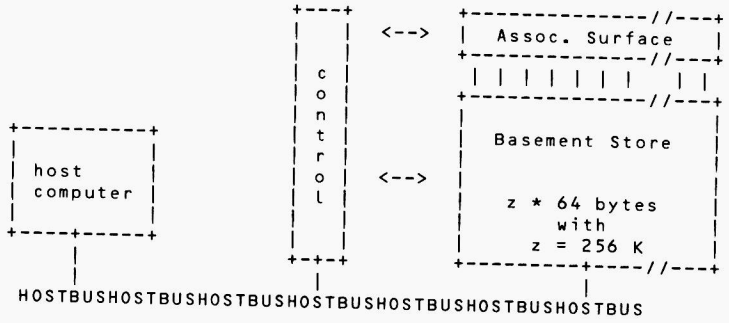

Fig. 2.1: computer system with HAS

HAS is a combination - a "hybrid" - of:

- A very large memory, the "Basement Store", organised as a 2-dimensional array with 64 columns and some large number of lines (e.g. 256000). Each HAS memory cell is of single-byte size. Text strings are usually written "top to bottom" into memory.

- 64 processor elements (PE), i.e. simple computing units. All PEs run synchronously and, at any single moment, they refer all to the same line of memory (called a "byte slice"). Due to this synchronous operation, HAS can perform certain byte array operations (e.g. certain searches) at an extremely high throughput. Every PE is directly linked to its respective memory column; an entire line of memory can thus be dealt with in one blow. In detail, every PE consists of an ALU-and-test unit (together with two registers A and B), a masking register M, and a bank of registers C; all are in some way connected to the memory bus (fig. 2.2). The ensemble of the PEs is called the "Associative Surface".

- A central control unit; it initiates each and every action in the PEs (i.e. it sends a succession of commands to the PEs). It can also access computation results of the PEs, and its further proceeding can be influenced by such findings. - The control unit contains among others a microprogram control unit and an address unit for the Basement Store.

HAS requires a host computer for its operation. The host computer looks after input and output, loads data into HAS, retrieves results back from it, and issues computation requests to it. Typically, HAS would be attached directly to the host bus: the HAS store would function as the host's main memory, or at least part of it, and the HAS control unit would behave like a DMA I/O device (with command and status registers, and with a provision for mutual interrupts).

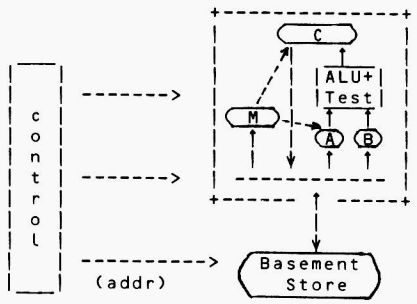

Fig. 2.2: one vertical slice of HAS

HAS is clearly a SIMD machine (array processor), and is as such ideal for the uniform processing of large volumes of data. A certain degree of flexibility is obtained by means of the masking register M.

The HAS concept can be summarised:

- The HAS hardware divides up into two main components, the Basement Store and the Associative Surface. Due to the modular design, the Basement Store can always be upgraded to the latest type of memory chips. Moreover, these chips can be bought off-the-shelf, thus greatly reducing its cost of manufacture.

- The Basement Store consists of dynamic RAM chips, i.e. the present standard technology. The Associative Surface, on the other hand, is built in a high performance technology (version 1: ECL, version 2: VLSI with pipelined command execution) which matches the speeds of Basement and Surface: the Surface can go through an entire cycle of computation while the Basement goes through one memory cycle. The memory chips are thus used at their maximum read/write rate.

The Associative Surface is the most decisive part of HAS: most of the processing takes place here, and most of the innovative ideas are also to be found here. Fig. 2.2 shows a highly simplified diagram of a single PE with its integration into HAS. On the left we see the control unit; it supervises all parts of HAS. The Basement Store BS (more precisely: its column relating to "this" PE) is shown at the bottom. Essentially, all data flow from BS to the ALU, its result flows on into a C register from where it can return into BS:

BS -> ALU -> C -> BS

The ALU-and-test unit can also perform the usual tests (equality, less, greater, overflow etc). The masking register M serves to inhibit the changing of any register.

The PEs are tailored for some particular algorithms (byte array operations, especially search and sorting), while many other algorithms may harmonise rather poorly with HAS. Searching a given string out of 64 candidate strings is probably the most "natural" HAS operation.

In the control unit, a microprogram determines the order in which to carry out the various actions (placing values in registers, setting switches, triggering computations). HAS is thus able to perform, upon one host instruction, even the most complex algorithms. An entire search command, for example, could be one such instruction. One should differentiate between instructions which can be concluded in a single Basement memory cycle, and those requiring more time. Among the former are:

- search for the occurrence of a given character,
- comparing the entire byte slice with a given character.

Instructions of the latter kind are necessary as soon as we need to handle more than 64 bytes. Such applications would be:

- comparing arbitrary strings,
- searching sets of strings,
- minimum, maximum, sorting, similarity,
- weighed search,
- join operation.

In cases where sorting is of prime importance, a significant speed-up can be achieved by a simple hardware extension (see [Wolf85]).

3. Standard Database Applications

The SIMD architecture makes HAS a powerful tool for set operations in database applications. In particular, operations known from the relational model such as selection, projection, and join can be performed efficiently through the parallelism offered by the 64 PEs.

3.1 Basic Database Operations

Let us assume that all relations to be processed have already been loaded into HAS. Each relation consists of one or more segments, a segment being a section of store one tuple deep and 64 bytes wide (see also fig. 3.1).

We introduce a simplified subset of HAS instructions with the intention of demonstrating how basic database operations can be processed on HAS. The six instructions are not claimed to describe HAS exhaustively and precisely.

(1) Uniform load b(i) into A
loads one byte from address i into the A registers of all PEs.

(2) Vector load sl(j) into B
loads one byte slice from BS line j into the corresponding registers B of the PEs. In contrast to (1), all B-registers contain in general different values after loading.

(3) Vector load A into B
loads, in each PE, the contents of register A into register B.

(4) Vector test for equality A and B
compares the contents of registers A and B of each PE. In case of equality, the corresponding result is set to one. The instruction delivers a vector as result.

(5) Vector store test result into C(k)
stores the result vector of a preceding instruction (4) into address k of the C register bank of the PE.

(6) Rotate A by one
Transfers the contents of each A register to the A register of its cyclic neighbour PE. In reality, this transfer is not achieved directly but via a bus connecting all PEs.

We shall now demonstrate, in terms of these six instructions, how the selection, projection, and join operation on a segment can be accelerated by the use of the parallelism offered by HAS.

The selection operation on a segment in HAS can be processed in the following way:

```
for  i = 0  to  (length-1)
   do
     uniform load  b(i+i0) into A
     vector  load sl(i+j0) into B
     vector  test for equality A and B
     vector  store test result in C(i+k0)
   od
```

We assume that the search argument and the segment have been loaded into Parameter Store (addresses i0 to i0+length-1) and Basement Store (addresses j0 to j0+length-1) respectively (fig. 3.1). Here, the uniform load instruction loads byte b(i+i0) of the search argument into all A registers. The vector load instruction moves the bytes of byte slice sl(i+j0) from BS to the corresponding B registers. After a test for equality has been performed in parallel for all PEs, the result vector is stored into address i+k0 of the C register bank. By "vertically" ANDing all (n=length) result vectors, the tuples equal to the search argument can be located.

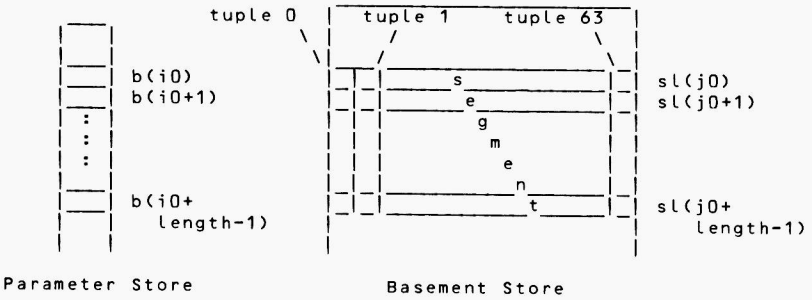

Fig. 3.1: layout of search argument and segment

The projection operation consists, in the first step, in cutting off domains. The second step, i.e. detecting duplicates among the tuples of the cut segment, can be accomplished by HAS in the following way:

```
for  i = 0  to  (length-1)
   do
     vector  load sl(i+i0) into A
     vector  load A into B
     for  j = 0  to  31
        do
          rotate A by one
          vector test for equality A and B
          vector store test result in C((j+k0)*(i+1))
        od
   od
```

The same byte slice sl(i+i0) is loaded into both PE registers. The byte slice residing in the A registers is rotated. After each step of rotation a test for equality is performed, and the result is stored in the C register bank. The stored results of the comparisons can be used to finally identify and eliminate all duplicates.

A join operation on 2 segments of different relations can be performed by HAS in the following way:

```
for  i = 0  to  (length-1)
   do
     vector  load sl(i+i0) into A
     vector  load sl(i+j0) into B
     for  j = 0  to  63
        do
          vector test for equality A and B
          vector store test result in C((j+k0)*(i+1))
          rotate A by one
        od
   od
```

The byte slices sl(i+i0) and sl(j+j0) from the first and second relation are respectively loaded into the registers A and B. Byte slice sl(i+i0) is stepwise rotated in registers A; after each step of rotation, a test for equality is performed. The result is kept in the C register bank for later evaluation.

3.2 Operations on the Application Level

As in most database machine concepts, the overall performance of HAS depends strongly on the load profile of the particular application.

Although a Basement Store capacity of some 16 Mbytes is envisaged for HAS, many applications require a much larger working set. This gives rise to the problem of fast mass data transfer between primary and secondary store, a problem not unknown from, but nevertheless unsolved in, typical database machine architectures (various solutions are discussed in [BoDe83]). Taking into consideration data rates of available mass stores, no device exists at present which can cope with the HAS data rate of about 80 Mbytes/sec. For the time being, we focus on applications which require only little external loading.

The parallelism provided by the SIMD architecture of HAS is well suited to support low-level set operations. Since strong synchronism is established between processor elements of the Associative Surface, only little communication is necessary during these operations. Parallelism on higher levels, however, cannot be easily achieved with this type of architecture.

Nowadays, many applications require on a low level single-record operations, thus prohibiting parallelism. It is not clear how often this results merely from the original database design where a discourse world was modelled for a database management system without set operations. A redesign of the original application by means of a conceptual model may in those cases lead to a structure more suitable for this type of database machine architecture.

3.3 Performance Considerations

The internal search speed attainable with HAS is only limited by the cycle time of memory chips. Nowadays, even memory chips of large capacity have cycle times of 500 ns or less. Since whole byte slices of 64 bytes are accessed and processed in parallel, search data rates of 128 Mbytes can be achieved.

4. Associative Processing in Deductive Databases

4.1 Deductive Databases and Prolog

Databases can be augmented by deduction rules serving to infer information not explicitly stored in the database. Although such "deductive databases" (DDB) can in principle be implemented in any language, we will in this section study a Prolog-like implementation. Our discussion will thus deal not only with DDBs but with the efficient implementation of the internal database of Prolog and logic programs in general.

We do not intend to explore the problems entailed in interfacing a Prolog system to a database residing on background store. This problem, though important, will be investigated separately. We will concentrate on the non-evaluational approach [Chan78], and assume that all relevant data have been brought from the disk-based database into the Prolog runtime environment beforehand. Since the standard depth-first execution strategy tends to be very inefficient, parallel execution of logic programs is being explored intensely by others [UmTa83].

Associative devices like HAS are ideally suited for synchronous operations on sets of data with regular structures. AND-parallelism in the evaluation of subgoals, on the other hand, requires independently running operations, which would be hard to implement on HAS; multiprocessor architectures are better suited for this purpose.

However, two important performance issues of Prolog implementations are amenable to associative processing (e.g. by a coprocessor of a DB machine): unification and OR-parallel execution of facts (i.e. a unit clause, a clause without body). Both problems are strongly interrelated. Unsuccessful unification of facts poses a major performance problem as has been shown in a measurement study of a Prolog system [Bull84]. The ratio of successful to unsuccessful unifications was reported to be roughly 1:6 in a DB-oriented program. We therefore concentrate subsequently on the enhancement of unification.

4.2 Implementation of Facts and Rules using the HAS

Due to the flexible loading mechanism of the HAS Basement Store, rules and facts can easily be stored either horizontally or vertically (fig. 4.1).

We assume functor and variable names as well as constants to be encoded into fixed length format. First, we will only deal with facts.

If facts are stored horizontally they occupy one or more 64-byte memory slices. Suppose, the clause

```
P(X,Y) :- Q(Z,X), R(Z,Y).                                    (*)
```

is to be evaluated and X has already been bound to the constant a. Let Q and R be facts. Many unsuccessful unifications are usually attempted on Q. Their number cannot be decreased by using HAS, but the execution time will be vastly reduced: for example, if we take a fact Q(<const>, <const>) and call it f,

```
f  :=  Q(<const>, <const>)
j  :=  number of bytes required to represent f in HAS
k  :=  entier( (j+63)/64 )
```

only k memory cycles are needed to match Q(Z,a) against f.

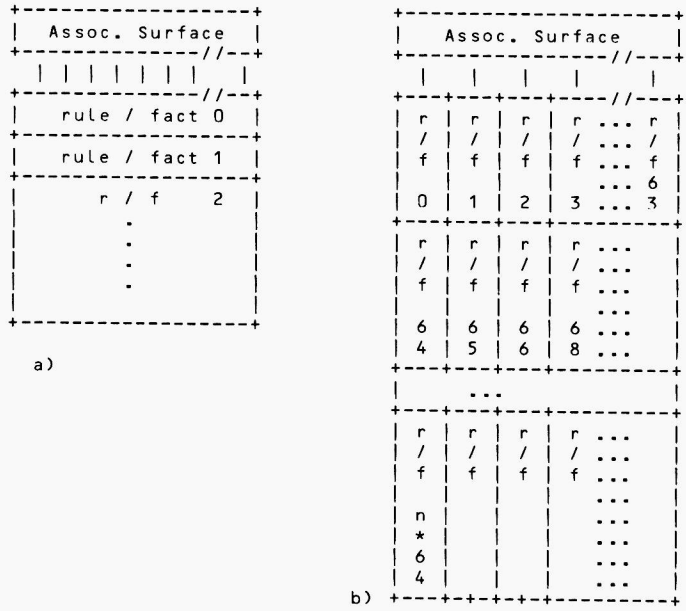

Fig. 4.1: rules/facts stored a) horizontally or
b) vertically in HAS

Furthermore, the set of all solutions for Q(Z,a) is generated in one blow, in contrast to the standard depth-first evaluation of Prolog. Unification of the above type is obviously only a special case: it corresponds directly to the selection operation of DB processing. It is implemented as an associative (horizontal) search, using the masking facility of HAS for the variable parts of the "query". Coming back to (*), we need to determine now those Y-values which correspond to one of the Z-values obtained in the previous step. However, this requires merely to join {b : Q(b,a)} with the facts given by R. The join operation has already been discussed in chapter 3.

If we store the facts vertically in the HAS memory, 64 facts can be unified with a goal statement in parallel. The goal is compared character by character with the clauses. Variable parts of the goal are masked, and replaced by constants, if the match succeeds. It should be remembered that we deal with a simple form of unification where a constant / constant check is carried out. This method (fig. 4.2) is very similar to searching among conventional relations.

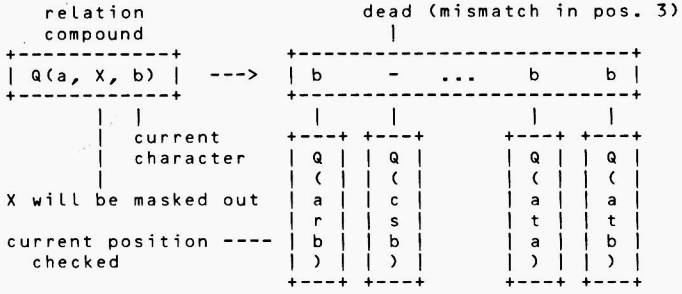

Fig. 4.2: parallel matching in HAS

All solutions to Q(...,X,...) are generated (OR-parallelism) just as in the case of the horizontally arranged clauses. Performance is primarily determined by the memory cycle time. Supposing each fact was stored in 200 bytes, and a 500 microsec RAM was used, it would take 1 msec to unify 640 facts (plus the time to read the variable bindings).

The situation gets more complicated when dealing with rules instead of simple facts. Rule bodies may be conjunctions of any number of predicates. If the defining clauses for Q have such bodies, synchronous reduction of a subgoal Q will work only as long as the bodies are conjunctions of the same predicates, e.g.:

```
Q(a,X) :- P(X,a,b).
Q(X,a) :- P(X,Y,a).
```

However, an associative device like HAS can be of advantage if the clauses are laid out in HAS memory in a special way. Let us symbolise each instance of a predicate in the set of rules and facts by an index <i,j> where i is the clause identifier and j is the position within clause i (<i,0> thus being the head of clause i). Facts and clauses (with any number of subgoals) can be uniformly represented in HAS memory through sequences of such an index put together with functor and arguments. The following example illustrates the method:

Prolog Program (including "database"):

```
line                          (In Prolog, atomic constants are
  1     D(h).                  denoted by lower case letters.)
  2     M(m,e).
  3     M(m,w).
  4     M(h,X) :- M(X,w).
  5     S(X,Y) :- D(X), M(X,Y).
  6     S(h,g).
        |         |     |
        0         1     2   (depth level)
```