Acceleration of Nonnumeric Operations Using Hardware Support for the Ordered Table Hashing Algorithms

Emil Jovanov, *Member*, *IEEE*, Veljko Milutinovic, *Senior Member*, *IEEE*, and Ali R. Hurson, *Senior Member*, *IEEE*

Abstract—This paper introduces a new approach to acceleration of nonnumeric, database, and information retrieval operations. While traditional techniques accelerate the most time-critical high-level software constructs, we propose novel low-level primitives and demonstrate how these primitives improve database operations. Radix sorting, hashing, and bit-vector operations are used to develop a new class of nonnumeric algorithms—**OTHER** (**O**rdered **T**able **H**ashing and **R**adix sort algorithms)—based on low-level hashing operations Init, Mark, and Scan. We have proposed and evaluated two hardware accelerators for OTHER algorithms. It is shown that a low complexity hardware support (less than 10K transistors) can significantly improve the performance of nonnumeric operations.

Index Terms—Database operations, hashing, sorting, searching, nonnumeric processing, hardware accelerators.

1 Introduction

Conventional computer systems, by their very nature, are sequential machines, supported by an arithmetic logic unit structured for numeric computations and a passive address-accessible memory hierarchy. The ability to process efficiently large amounts of nonnumeric data is crucial for many computer applications, such as database and information retrieval. They are characterized by simple, repetitive nonnumeric operations on a massive volume of data where, in general, data locality is not preserved. This incompatibility has resulted in the so-called semantic gap, computation gap, and size gap [1].

The challenge to reduce the aforementioned gaps has motivated a great deal of research since the mid 1970s, e.g., database machines (DBM) [1]. Using the processor architecture and database functionality as classification taxonomies, one can distinguish three classes of database machines: application-specific DBMs, general-purpose DBMs, and general-purpose computers with increased database performance. However, it has been shown that suitable database performance can be easily achieved using VLSI accelerators [2], [3], [4], [5], [6], [7]. The approach proposed in this paper is based on hardware acceleration of general purpose CPUs.

A great deal of research in the field of database machines has focused on the development of dedicated database architectures [1], [8], [9], [10]. Parallel to this work, some vendors have developed general purpose machines with database functions such as *select*, *search*, and *join* integrated directly into the machine architecture [11]. In spite of the commercial success of database machines (hardware or software-based organizations), the generality of the von Neumann architecture has also motivated another approach to the efficient handling of database systems, i.e., machine-level instructional support for operations that can improve the performance of the database operations (e.g., bit string manipulation instructions). For example, Intel i386/486 processors support bit manipulation instructions [12] that can be used to implement primitive database operations [13]. Interestingly, the Teradata database machine also uses the very same set of operations to facilitate the efficient execution of database functions [14]. Our simulation results show that, even in the case of a manually optimized assembly program based on these dedicated instructions, acceleration of complex bit-manipulation operations is 1.5 to 3 times compared to an equivalent C program [13]. However, in practice, the performance improvement for database functions is much less.

Advances in VLSI technology would suggest an alternative approach to improve performance of the database functions, namely, VLSI accelerators. Most of the realized VLSI accelerators for the database environment are intended to accelerate high-level functions, such as select [9], sort [3], [6], and join [1]. This paper proposes an alternative to this direction; simply put, instead of accelerating high-level operations, we accelerate "lower-level" operations (see Section 2.1). Conventional RISC instruction set optimization is generally based on statistics of (mostly numeric) benchmarks implemented using an existing instruction set. Therefore, in most cases, it represents optimization of the *existing* instruction set. Nevertheless, a different set of basic operations generates completely different execution statistics for

- *E. Jovanov is with the Electrical and Computer Engineering Department, University of Alabama in Huntsville, Huntsville, AL 35899. E-mail: jovanov@ece.uah.edu.*
- *V. Milutinovic is with the School of Electrical Engineering, University of Belgrade, 11000 Belgrade, Yugoslavia.*
- *A.R. Hurson is with the Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16802.*

Manuscript received 2 Oct. 1999; revised 5 Dec. 2000; accepted 18 Mar. 2002. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 110694.

TABLE 1
Implementation Hierarchy of Database Operations

Procedure Level	Operations
High – HL	Database operations (select, ordered select, sort, join)
Medium – ML	Ordered table hashing operations (E-class processing)
Low – LL	Low-level ordered hashing primitives (*Init, Mark, Scan*)

the given application. Since the design of a dedicated instruction set can hardly be economically justified, we propose a standard, general-purpose processor core communicating with a low-level operation accelerator. The proposed Low Level Accelerator (LLA) can be implemented either as an on-chip resource or an off-chip coprocessor. Although the LLA concept is not new, to the best of our knowledge, it has not been used extensively for nonnumeric database operations. Order hashing primitives resolve the semantic gap by offering a mechanism to directly position every processed object in the approximate location in the final result set. Efficient processing of modified algorithms allows processing of larger data sets in shorter time, providing a cost-efficient solution for computation and size gap problems.

Section 2 addresses our general approach and its mathematical foundation. In addition, it introduces our basic algorithms. The set of proposed algorithms, introduced in Section 2, is extended for hardware acceleration. Analytical modeling of *select*, *sort*, and *join* database operations is discussed in Section 4. The design of an accelerator for the proposed primitive operations is introduced in Section 5. Section 6 presents the simulation results and, finally, Section 7 concludes the paper.

2 Low-Level Acceleration

This paper presents optimization of the cost/performance ratio of nonnumeric database operations based on the low-level ordered hashing primitives. Implementation hierarchy is presented in Table 1. Standard database operations are implemented using ordered table hashing operations that rely on low-level hashing primitives. In the modified algorithms, we observe some suboptimal low-level operations that offer a higher speedup through on-chip acceleration. The relative share of low-level table processing operations is up to 90 percent, as presented in Section 6 (Table 3). Consequently, we focus on an unorthodox research direction: introduce a new set of primitive operations, accelerate the most critical operations in this new set of primitive operations, synthesize high-level functions, and introduce modified algorithms for basic database functions based on this new set of primitive operations.

The proposed set of primitive operations is called **OTHER** (**O**rder **T**able **H**ashing and **R**adix sort) since it is heavily based on a hashing technique for database operations. We implement the most frequently used database operations, *select*, *sort*, and *join*, using ordered table hashing, based on bit vector operations for effective table processing. The low, medium, and high-level operations are described in Sections 2.1, 2.2, and 2.3, respectively.

2.1 Low-Level Operations/Table Manipulation Primitives

We decided to develop the **OTHER** algorithms based on the order-preserving hashing technique because of the ability of hashing to reduce the search space and, hence, to derive more efficiency [15]. Commonly, the results of select database operations must be sorted. Therefore, instead of scattering the records across the hashing table, we use the order-hashing functions and the Address Calculation Sort Method [3], [16], [17] to order the records in a logical fashion in the hash table. Finally, the collision processing overhead among the duplicate and synonym keys (records with different key values, generating the same hash value) is reduced by using the logic identifier of each record. The logic identifier uniquely represents each record based on its relative position within the processed data set. This allows a unique correspondence between the record position and its position in collision sets [18], [19], [20]. Moreover, instead of maintaining a separate pointer in the list of collisions [3], the logic identifier itself can be used as a pointer to the next member of the list of collisions. In the proposed algorithms, the logic identifier is implemented simply as a record counter during the initial processing phase. Table 2 summarizes the notation used in this section.

Formally, assume we have an unordered set of N objects A

$$A = \{a_i \mid 1 \le i \le N\} \tag{1}$$

and an ordered hashing function ϑ generating values in a domain D_2 with maximum cardinality M,

$$\vartheta(a_i) = \{\, \vartheta_j \mid 1 \le j \le M\}. \tag{2}$$

The hash function ϑ preserves the order of objects in set A such that, for every pair $\{a_i, a_j\}$ and a given relation ρ between objects, we have

$$a_i \,\rho\, a_j \;\Rightarrow\; \vartheta(a_i)\rho\, \vartheta(a_j),\ \rho \in \{<, =, >\}, i \ne j,\ 1 \le i, j \le N. \tag{3}$$

Let **S** be an ordered hashing table

$$S = \{s_i \mid 0 \le i \le q, 1 \le q \le M\} \tag{4}$$

and **C** be a set of synonyms

$$C = \{c_i \mid 0 \le i \le N_c, \quad 0 \le N_c \le N\}. \tag{5}$$

Members of sets **S** and **C** can be object identifiers or pointers to objects. We consider each record to be identified by its relative position within a block of records in the main memory. As a result, sets **S** and **C** can be realized as vectors of logic identifiers and, therefore, s_i and c_i are equivalent to $S[i]$ and $C[i]$, respectively. This approach allows efficient computer implementation using sort array **S** and collision array **C**. For example, if the objects a_i represent numbers,

TABLE 2
Notations and Definitions

Term	Definition
A	Set of objects a to be processed (for database operations – attribute values)
N	Number of objects (cardinality of set A)
D_1	Domain of values of attribute A
ϑ	an ordered hashing function
D_2	Domain of values of the hashing function ϑ
Q	Number of unique hash values of the function υ in set A (cardinality of D_2)
N_c	Number of collisions (synonyms)
S	The ordered hashing table
M	Size of the ordered hashing table (maximum cardinality of D_2)
C	The collision table
L	number of key partitions processed
λ	Special symbol denoting an empty table entry
E_j	*E-class*, set of all objects with the same value ϑ_j of the hashing function ϑ
Ψ	Set of headers of *E-classes*, the result of *Scan* operation

according to (2), a trivial implementation of the hashing function is to use $\vartheta(a_i) = a_i$ and i as the object identifier. Therefore, a_i is used as an index of table entry ($\mathbf{S}[a_i]$), where the object identifier i is written, while all other synonyms with the same hash value are written into collision table **C**. An example of this implementation is provided in Section 3.3. However, if the object a_i is represented using m bits, we need an ordered hashing table **S** of 2^m entries and the collision table **C** of N_c entries.

The ordering process requires two basic operations for every processed object:

- *Mark*, to write the object identifier into the ordered hashing table **S** and link synonyms with the same hash value using collision table **C**.
- *Scan*, to retrieve the next object identifier in the sorted order.

Successive calls of operation *Scan* generate a set of object identifiers with hash values between given limits in the sorted order:

$$\Psi = \{S[\vartheta_j] \mid LowLimit \le \vartheta_j \le HighLimit, S[\vartheta_j] \ne \lambda\}. \quad (6)$$

We use three low-level primitive operations in the proposed middle and high-level algorithms:

- *TableInit* initializes the entries in the hash table by means of "null pointers" (Algorithm *LL-1*).
- *Mark* operation $Mark(\vartheta, a, i)$ performs hashing of the *i*th object (a_i) using hash function $\vartheta(a, i)$. Marking is a three-phase process: First, the hash value of the current partition of the key ($\vartheta(a_i)$) is calculated, then the logic identifier of the key (*i*) is written into the hash table **S**, and, finally, the logic identifier is linked into the list of synonyms **C** (Algorithm *LL-2*). In our research, the hashing function is implemented in software to extract $log\, M$ bits from the processed key. Additional acceleration and a smaller number of collisions can be achieved using a hardware-based hashing function [21]. Programmable hashing function can be easily integrated as a part of the accelerator.
- *Scan* operations generate the next ordered identifier from the hashing table, satisfying the condition that the hash value is between the given limits. Two types of scan functions are distinguished: *GetFirst* generates the identifier of the first key larger than a given partition value and *GetNext* continues the search from the current table position (Algorithms *LL-3* and *LL-4*, respectively).

Algorithm LL-1. ***TableInit***

```
TableInit() {
    for (i=0; i < M; i++) {
        /* Write "null pointer" to every table entry */
        S[i]=λ;
        }
```

Algorithm LL-2. ***Mark***

```
Mark(ϑ, a, i); {
    /* Calculate hash value of the current partition of
       the key */
    p = ϑ(a);
    if ( S[p] == λ ) { /* Table entry is free */
    /* record is the only member of E-class, write
       "end_of_list" to table C */
           C[i] = λ;
           S[p] = i; }
    else { /* Table entry is occupied */
    /* link previous identifier from the hash table
       in the collision table */
           C[i] = S[p];
           S[p] = i; }
    }
```

Algorithm LL-3. ***GetFirst***

```
GetFirst(StartScan) {
    i=StartScan;
    while ((S[i] == λ) && (i <= M)) {
           /* Search for the first nonempty table entry */
           i++;
           if (i <= M)
                  /* Nonempty entry found */
                  return(i);
```

```
            else
                        /* End of scan */
                        return(λ);
            }
}
```

Algorithm LL-4. ***GetNext***

```
GetNext(i, EndScan) {
    while ((S[i] == λ) && (i <= EndScan)) {
            /* search for nonempty table entry */
               i++;
            if (i <= EndScan)
                        /* Nonempty entry found */
                        return(i);
            else
                        /* End of scan */
                        return(λ);
            }
}
```

2.2 Ordered Table Hash Algorithms

Low-level primitives *TableInit*, *Mark*, and *Scan* perform ordered hashing of individual objects (database attribute values). Medium-level procedures comprise the low-level procedures and perform ordering of all objects in input set **A**. The most important problem of hash-based algorithms is resolution of synonyms [22]. Collisions are linked to comprise a set called the equivalence class or *E-class*.

Definition 1. *The equivalence class or* E-class E_j *is the collection of* a_i *such that*

$$E_j = \{a_i \mid \vartheta(a_i) = \vartheta_j\},$$

where

$$\vartheta_j < \vartheta_{j+1}, \quad 1 \le j < q, \;\; 1 \le q \le M.$$

All members of the equivalence class must be processed to resolve collisions and make final order within the class. This is usually performed in software using algorithms with fast sorting of small sets, such as the List Insertion Sort method (LIS) [22]. The equivalence class consisting of only one object is called final and requires no further processing. Based on the key size and available main memory, the order-preserving hashing function can be applied to the whole key or a key partition for partial ordering. Similarly to radix sorting, the sorting starts from the most significant position of the key. Keys are sorted by recursive use of ordered hashing and the LIS sort. Partitions involved in radix sort are not always taken from the absolute start of the key; rather, we can start from the most distinctive part of the keys.

The efficiency of hash-based algorithms critically depends on the quality of the hash function and the number of collisions it generates. The worst-case performance, when all the keys are hashed into the same table entry, traditionally creates fear of hashing. It has been shown that the hash function can achieve analytical performance with real-life data [21]. If the hash function generates all hash values with equal probability, then the number of collisions can be approximated using Poisson distribution [23]. The number of single member *E-classes* (final classes without collisions) N_f after processing N keys using a table of size M is

$$N_f = N\, e^{-N/M}, \tag{7}$$

while the number of occupied entries in hash table N_m will be

$$N_m = M(1 - e^{-N/M}). \tag{8}$$

According to (7) and (8), when a large hash table is used ($M >> N$), almost all *E-classes* are final ($N_f \approx N$). This means that the result of the initial sorting phase contains a sorted list with a small number of collisions, requiring further sorting. On the other hand, for small hash tables ($M << N$), N_f is small and the average problem size is reduced from N to N/M. This is particularly important for operations with nonlinear execution time, such as sorting (with $O(NlogN)$ complexity). In conclusion, a large hash table decreases problem size and collision processing time at the expense of increased table processing time and larger memory requirements.

OTHER algorithms link members of *E-classes* using the logic identifier of each record. In this way (similar to the techniques proposed in [22], [24]), there is no need to maintain additional pointers. Consequently, according to *Algorithm LL-2*, the equivalence class could be defined recursively as

$$\begin{aligned} &E_j = \{S[j],\, C[S[j]],\, C[C[S[j]]], \ldots\}, \\ &S[j],\, C[i] \neq \lambda, \quad i = 1..N. \end{aligned} \tag{9}$$

The first element of the E_j (S[j]) is called the *header*. This is the only element stored in hash table **S** and it is used to access all the other members of the *E-class* stored and linked in the collision table. According to Definition 1 and (3), the set of the equivalence classes is also ordered. Therefore, let us define an ordering function *F* that generates an ordered set of *E-classes*:

$$F(A, \vartheta) = (E_1, E_2, \ldots, E_j, \ldots, E_q), \quad 1 \le q \le M. \tag{10}$$

If the hashing function cannot be applied to the whole key value, the key values can be partitioned and the aforementioned hash-based sort algorithm can be applied, recursively. Formally, let us assume the following partitioning of key a_i into L partitions:

$$a_i = a_{i1}\, a_{i2} \ldots a_{il} \ldots a_{iL}\,, \qquad l = 1..L. \tag{11}$$

When partitioning is used, a low-level *Mark* operation is applied to the current key partition: $Mark(\vartheta, a, i, k)$ performing hashing on the *k*th partition of the *i*th key (a_{ik}) using hash function ϑ. Ordering function F is applied recursively:

$$F(A,\, \vartheta(a_i)) = F(F(F(A,\, \vartheta(a_{i1})), \vartheta(a_{i2})), \ldots, \vartheta(a_{iL})). \tag{12}$$

It can be seen that, after *l* partitions, even for a small hash table, the average problem size is reduced at least from N to n_l, where n_l is the average length of list of synonym keys. After *l* partitions, the average *E-class* length n_l is equal to

$$n_l = \frac{N}{M^l}, \qquad l = 1..L. \tag{13}$$

In the case of sorting, according to (13), the complexity of the algorithm $O(NlogN)$ is decreased to $O(N) + O(n_l\ log(n_l))$, where n_l is the average *E-class* length after processing l partitions. In this case, even a suboptimal algorithm, such as List Insertion Sort (LIS) [22] can be used to sort synonym keys. Our measurements on the DEC Alpha workstation indicate that the LIS is the most efficient solution for sorting *E-classes* of up to 27 collisions.

Algorithm *ML-1* presents the general *E-class* processing procedure. It orders all members of the *E-class* with the *k*th key partition between $LowLimit_k$ and $HighLimit_k$. This is the major medium-level building block for the **OTHER** algorithms. The result of the *Scan* operation (set Ψ) is pushed on the stack for further processing. For example, the ordered select algorithm (Algorithm *ML-2*) will repeatedly call on the *E-class* processing. The ordered select procedure is composed of three major blocks:

- The initial loading performs the highest level ordering and creates an initial set of *E-classes* from the original set $(\mathrm{F}(\mathrm{A}, \vartheta(a_{i1}))$.
- The second phase orders the set of *E-classes*, satisfying the selection criterion. For optimal acceleration, only *Lopt* partitions are processed and single member *E-classes* are placed directly to the result stack. We can represent this as $\mathrm{F}(\mathrm{A}, \vartheta(a_{il}))$, $2 \leq l \leq Lopt$.
- Finally, in the third phase, the remaining short lists are processed online during the creation of the final result.

Algorithm ML-1. ***E-class processing***

```
int Process_E_class(Header,k) {
    /* Initialize the hashing table to process kth
       partition of key */
    TableInit();
    /* Start from the E-class header */
    i = Header;
    do {
        /* Hash and link kth partition of ith key */
        Mark(ϑ, a, i, k);
        /* Get the next member of E-class, linked in
           the collision table */
        i = C[ i ];
    } while (i ! = λ );/* for all members of E-class */
    i = GetFirst(LowLimit_k);
    /* Find the first key larger than kth partition of
       LowLimit in the hashing table */
    if (i == λ) then
        /* Return "nothing to select" status */
        return(1);
    else {
        /* Push i on the result stack */
        push( i );
        do {
        /* Search the hashing table for the next
           member of E-class, smaller than kth
           partition of HighLimit */
            i = GetNext(i, HighLimit_k);
            /* Push i on the result stack */
            push( i );
        } while (i ! = λ );/* for all members of E-class */
        return(0) ;          /* Return OK status */
        }
}
```

Algorithm ML-2. ***Ordered Select***

```
int OrderedSelect(LowLimit, HighLimit) {
/*** 1. Initial loading */
    /* Initialize the hashing table to process the set as
       the initial E-class */
    TableInit();
    /* Hash and link the first partition of ith key */
    for (i=0; i<N; i++)
        Mark(ϑ, a, i, 1);
    /* Order keys starting from the lowest value */
    i = GetFirst(LowLimit_1);
    if ((i == λ) | | (i > HighLimit))
        /* Return "nothing to select" status */
        return(1);
    push_TR( i );
    /* Push i on the temporary result stack */
    push_TR( i )
    /* Scan the hashing table */
    do {
        /* Search for the next member of E-class */
        i = GetNext(i, HighLimit_1);
        /* Push i on the temporary result stack */
        push_TR( i );
    } while (i ! = λ);   /* For all members of the E-class */

/*** 2. Ordered hashing is used to process
  optimal number of partitions - Lopt */
    /* Partition processing */
    for (k=2; k<=Lopt; k++) {
        do {
        /* Pop i from the temporary result stack */
            pop_TR( i );
            if (C[i] == λ ) then
                /* Final E-class, no duplicates
                   push identifier on the final
                   result stack */
                push_FR(i);
            else
                /* Process E-class */
                Process_E_class(i,k);
        } while(i ! = λ);
    }

/*** 3. Process unprocessed E-classes */
    do {
        /* Pop i from the temporary result stack */
        pop_TR( i );
        if (C[i] == λ )
            /* Final E-class, no duplicates push
               identifier on the final result stack */
            push_FR(i);
        else
```

```
                /* Process E-class using the List
                   Insertion Sort algorithm */
                Software_E_class_processing(i);
    } while(i != λ );
    return(0);        /* Return OK status */
}
```

2.3 Database Operations

We present implementation of the most frequently used database operations using ordered table hashing operations. Without loss of generality, we assume a relational database model [25]. For example, consider the relation schema:

Employee(EmployeeID, FirstName, LastName, HireDate, Salary)
Invoice(Invoice, CustomerID, EmployeeID, Date, Amount)

Database queries are declared using standard query language SQL [26]. We provide here examples of *select*, *multiple select*, *sort*, and *join* database operations. The *select* database operation selects a set of objects satisfying given conditions. Very often, the output of the select operation must be ordered on the value of some attribute. For example, selecting and sorting all employees hired after 1 January 1999, could be performed with the following SQL query:

Query 1:
```
SELECT LastName, FirstName, HireDate
FROM Employees
WHERE HireDate > #1-JAN-1999#
ORDER BY HireDate;
```

Sometimes *select* must satisfy multiple criteria. In the first example, Query 1 can be changed to find all employees hired after 1 January 1999, with a salary less than $45,000.00. In that case, the WHERE clause will be changed as follows:

```
WHERE HireDate > #1-JAN-1999# AND Salary<45,000.00
```

Another important database operation is joining information from two relations. This operation is called *join*. For example, Query 2 represents selection of all invoices made by employee Smith, ordered by date.

Query 2:
```
SELECT Employees.LastName, Invoice.Date,
Invoice.Amount
FROM Employees INNER JOIN Invoice ON
Employees.EmployeeID = Invoice.EmployeeID
WHERE Employees.LastName='Smith'
ORDER BY Invoice.Amount
```

A search for a specific record is accelerated using an index, although use of an index introduces an additional processing overhead. We use an ordered table hashing to select and sort fields without building an index over attribute values.

The generality of the **OTHER** algorithms allows us to decompose complex database operations into a set of primitive operations. As indicated earlier, hashing is often used to perform this decomposition. For the remainder of this paper (for the sake of simplicity without loss of generality), we assume an ideal hashing function with uniform distribution of hash values that can be approximated by the Poisson distribution. It should be noted that, in the proposed algorithms, the single element *E-classes* do not require further processing. Therefore, it is desirable to generate as many single element *E-classes* as possible as early as possible during the course of operations. Moreover, note that, in the final stage, the proposed algorithms simply process many short lists of elements, regardless of the size of the original problem. In other words, our algorithms decompose the original problem into a set of smaller problems determined by the number of collisions made by the hashing function. Problem decomposition is significant for operations with nonlinear execution time, such as sorting.

The *E-class* processing algorithm can be used to synthesize most of the database operations and even introduce some additional optimization. We present the most important applications in the following subsections.

2.3.1 Select

The *select* database operation is performed as an ordered selection between limits **Select**(*LowLimit*, *HighLimit*). Two types of selection operations are considered: 1) *SelectUnique*, where records with keys equal to a predefined value are selected, and 2) *SelectRange*, where a list of records with keys in a predefined range of values are generated. Naturally, conventional hash-based algorithms can be used to perform the *SelectUnique* function. For the *SelectRange* function, after the initialization and creation of the hash table, we select only the records with keys within the specified range during the scan phase. For example, Query 1 can be implemented using an *ordered select* (Algorithm ML-2) on the key HireDate as *Select*(#1-JAN-1999#, ∞).

Unfortunately, long keys require prohibitively large tables. Therefore, as mentioned earlier, keys are divided into L partitions and the above sequence of operations is performed, recursively, on generated *E-classes* within the generated range. The following results are the natural by-products of the *Select* operation:

- The operation generates a sorted (partially sorted) result. This could accelerate the execution of the other operations (e.g., aggregation) in the query. In most queries (such as Query 1), it is a desirable feature.
- The result of a select operation can be generated by using a few key partitions.
- The same initial hash table can be used to generate the results for multiple select operations. For example, in the modified Query 1, we can apply an ordered selection *Select*(0, 45,000.00) on the result of *Select*(#1-JAN-1999#, ∞).

2.3.2 Minimum/Maximum

The minimum and maximum values in the processed set are obtained by processing only the lowest/highest-order *E-classes* on every key partition, respectively. The modified *E-class* processing algorithm uses only the result of *GetFirst*(0) in ascending order for the minimum and *GetFirst*(M) in descending order for the maximum, for every key partition a_{il}, $1 \leq l \leq L$.

2.3.3 Sorting

Sorting is performed as an ordered selection in the whole domain of hash values D_2—$\mathbf{Select}(0, M-1)$ on every key partition. Formally, it can be represented as $F(HireDate, \vartheta)$.

The sort operation can be accelerated in two ways:

- Accelerate sorting by generating the *E-class* based on the most significant partition of the keys and then applying a traditional sort algorithm on the elements of each *E-class* or
- Recursively, apply the *E-class* algorithm on the members of the generated *E-classes* based on various key partitions.

2.3.4 Duplicate Elimination

If we execute *E-class* processing on all partitions of the key, all objects in the collision table will be duplicates. Therefore, set Ψ retrieved in the last *Scan* operation represents the set of objects without duplicate values.

2.3.5 Join

General Θ *join*, where $\{<, \leq, =, \geq, >\}$, is performed as an ordered selection over both relations. Joining of the two relations $\mathbf{A}$ and $\mathbf{B}$ can be represented as $F(\mathbf{A} \cup \mathbf{B}, \vartheta)$.

3 Extending the OTHER Algorithms Using Bit Vector Primitives

Bit vectors are often used to perform or accelerate nonnumeric operations [8], [27]. Furthermore, the literature has also addressed the application of bit vector operations in distributed systems as a means of reducing the communication cost [28]. As a consequence, the **OTHER** algorithms are extended by the application of bit vectors as a processing aid to improve performance.

The set of values generated by the hashing function ϑ on objects a_i, $\vartheta(a_i) = \vartheta_i$ can be represented using a bit vector:

$$c_0, c_1, \ldots, c_j, \ldots, c_{M-1}; \quad c_j \in \{0, 1\} \quad \forall\, 0 \leq j \leq M-1, \tag{16}$$

where

$$c_j = 1, \forall\, 0 \leq j \leq M-1, \text{ if and only if } \vartheta_i = j.$$

Elements of a bit vector can be accessed in two modes: *address access* and *associative access*. The basic **OTHER** algorithms can be extended by bit vectors and bit vector operations. For example, the hash table is assumed to have an associated bit vector of the same cardinality. Each entry in the hash table has a corresponding bit in the bit vector that represents its status (*i*th bit =1 indicates that the *i*th entry in the hash table is occupied). This allows the entries in the hash table to be initialized and scanned rapidly.

The ability of bit vector operations to improve performance also allows one to process larger hash tables. This further reduces the number of collisions and the cardinality of collision sets and, hence, accelerates the proposed algorithms. Bit vectors can be organized and operated as a flat file or, alternatively, as a hierarchical structure. In the hierarchical structure, a *W*-ary tree structure is organized to allow fast associative access. This organization is of particular interest when one is dealing with large, sparse bit vectors.

3.1 Flat Bit Vector Organization

In this organization, a bit vector of *N* bits is realized as a collection of $\lceil N/W \rceil$ words of length *W*. Each bit *b* is then referenced by a word number $\mathrm{C_b} = \lfloor \mathrm{b/W} \rfloor$ and a relative displacement p within $\mathrm{C_b}$, where $\mathrm{p = b - (\lfloor b/W \rfloor^* W)}$. The density of a bit vector is defined as $f_n = n/N$, where n is the number of marked bits in a bit vector of length N. Associative access to the flat bit vector may require, in the worst case, $N_{acc} = C_b$ number of accesses. For a small f_n, the bit vector scan is inefficient. As a result, we introduce a hierarchical organization to represent and access the bit vector.

3.2 Hierarchical Bit Vector Organization

In the case of sparse bit vectors, the hierarchical organization allows fast associative access. It is organized as a *W*-ary tree over the original bit vector. The height of the tree is defined as

$$LMax = \lceil \log_W N \rceil. \tag{17}$$

The fast associative access comes at the expense of more memory space. The total amount of memory needed to represent the hierarchical structure is

$$C_b^h = \sum_{i=0}^{LMax} C_b^i = \frac{W^{LMax} - 1}{W - 1}. \tag{18}$$

And, the maximum number of memory references needed for an associative access is

$$N_{acc}^h = 2 \cdot (LMax - 1). \tag{19}$$

It should be noted that the hierarchical organization requires $(LMax - 1)$ more memory references during the *Mark* phase operation.

3.3 Example

Fig. 1 depicts an example of the **OTHER** sorting process on a database where age is used as the key attribute. A hash table $\mathbf{S}$ of size $M = 10$ with its associated bit vector $\mathbf{BV}$ is assumed. The key is partitioned into two parts. The order-preserving hashing function, ϑ, in this case simply transforms a decimal digit from ASCII to its binary equivalence. The logic identifier represents the relative record position in the main memory, implemented as a counter. During initial table loading, logic identifiers are written in table entries designated by the first key partition; for instance, logic identifier 0 is written in table entry 6 and its corresponding bit in the bit vector is set to indicate that the cell is occupied. Note that each entry in the hash table ($\mathbf{S}$) is set to point to a possible linked list of *E-class* members in table $\mathbf{C}$ and a special value λ indicates the end of an *E-class* list. Fig. 1 shows the contents of the hash table, bit vector, and collision table after the initial loading. Application of a scan operation on the hash table $\mathbf{S}$ will generate a partial ordered list of identifiers [4, 3, 5, 0], where identifier 4 represents the head of the *E-class* [4, 2, 1]. Finally, the application of our hashing technique on generated *E-classes* based on the second key partition will produce the final sorted result list.

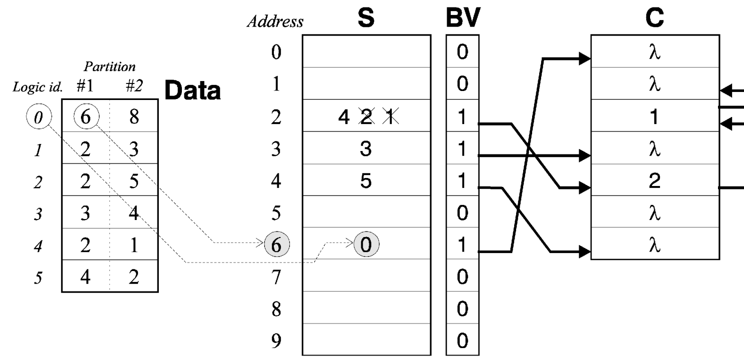

Fig. 1. Application of the proposed **OTHER** sorting algorithm.

4 Analytical Modeling

Analytical modeling is performed to demonstrate the acceleration of database functions when the **OTHER** algorithms are employed. We assume that the basic primitives are implemented in the hardware and the high-level operations are synthesized based on the table and bit-manipulation primitives defined earlier. The final result is a list of tuple identifiers (*tid*) without projection and physical relocation of tuples. Theoretically, maximum acceleration of the proposed algorithms are given as "**Z**ero time" **T**able **P**rocessing (**ZTP**) performance. It is the performance of an accelerator with infinite speed and private memory which does not require additional system bus cycles. Therefore, every read operation would find the result readily in the accelerator. This section discusses performance improvement of bit vector operations, as well as *Select*, *Sort*, and *Join* operations.

4.1 Bit Vector Operations

Fig. 2 gives possible acceleration of bit vector operations for the vector size of 1Mbit. Operations include vector initialization, mark, and scan primitives. As can be seen, hierarchical bit vector organization is more efficient than flat organization if the table utilization is less than 1 percent. Fig. 3 presents performance improvement. Conventional processors perform bit-vector scans as a sequence of shift-and-test steps, even in the case of dedicated bit operation instructions (Intel's X86 and Motorola's 680X0 families). Therefore, a software algorithm efficiently skips the empty words of a bit vector, but must scan the whole selected word bit by bit. The proposed accelerator performs single-cycle bit scan operations within a processor word. As a consequence, accelerators are the most efficient for a moderate table load factor, when every word in the bit vector has only a few bits marked. As can be seen from Fig. 3, for both organizations, acceleration depends on hash-table utilization. In addition, in both cases, certain values of load factors offer maximum acceleration.

4.2 Select

Fig. 4 shows possible acceleration of the *SelectUnique* for different key partition lengths (M) and the selectivity factor of 1 percent, key length is assumed to be 32 bits. As expected, performance of the select operation is heavily dependent on the selectivity factor. Even for a ZTP operation, performance improves only 20 to 40 percent relative to a traditional implementation.

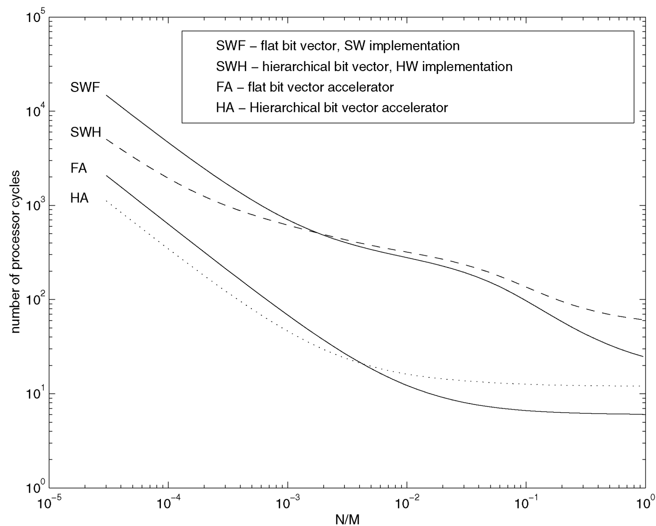

Fig. 2. Average number of processor cycles per key for bit vector operations (*Init*, *Mark*, and *Scan*) as a function of table utilization.

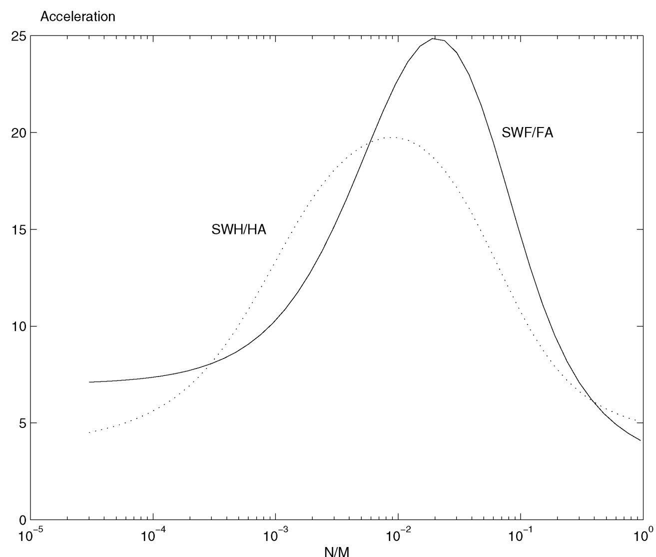

Fig. 3. Accelerator performance improvement for table operations for flat (**SWF/FA**) and hierarchical (**SWH/HA**) bit vector organization as a function of table utilization.

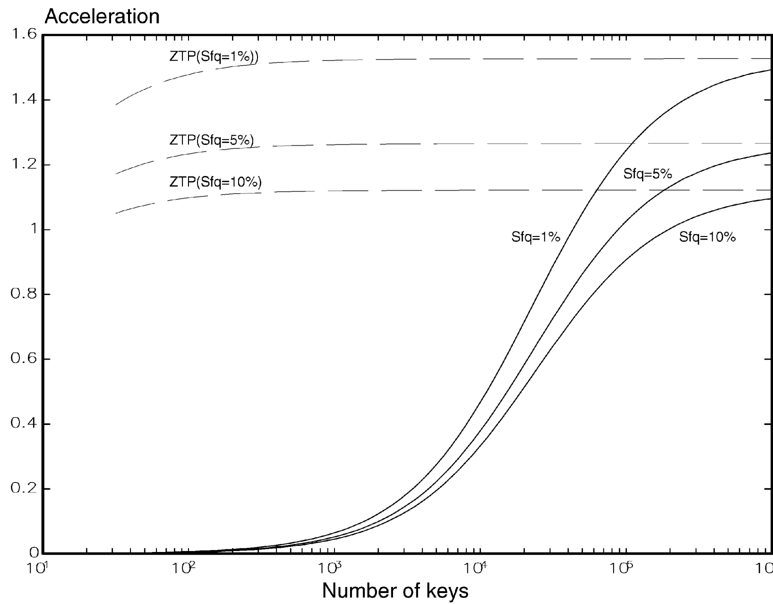

Fig. 4. Acceleration of *Select* operation for varying selectivity factors.

4.3 Sort

Fig. 5 depicts the theoretical acceleration of the Quicksort [22] in the case of recursive E-class processing and different key partition lengths. Increasing the cardinality of the database increases the number of collisions and, hence, increases the size of *E-classes* and, consequently, decreases the efficiency of their processing. Use of smaller partitions creates smaller *E-classes* in subsequent passes. Theoretically, infinite acceleration of low-level operations results in sharp "jumps" in performance, as it can be seen in Fig. 5.

Our analytical modeling also indicates the following:

- The proposed sort method has time complexity $O(N)$ if the partition domain size is equal to the sorting table size. The time complexity of the traditional sorting techniques is $O(NlogN)$. However, for a smaller table size ($M << N$) and application of a List Insertion Sort (LIS) to process the final lists, the complexity of the algorithm increases due to the larger complexity of the LIS algorithm. If l partitions are used in table-based preprocessing, according to (15), the algorithm complexity is still $O(N) + O(n_l^2)$, where n_l is the average *E-class* length.
- One can find an optimal table size for an underlying database that is directly dependent on cardinality (*N*).
- Traditionally, hash-based algorithms can suffer from table hot spots due to uneven distribution of hash values-data skew. As a result, longer *E-classes* can be generated. However, table-based preprocessing of the **OTHER** sorting algorithm determines exactly the length of every *E-class*. Therefore, the optimal sorting algorithm could be applied to sort collisions. Moreover, the low table processing overhead (see Table 1) will not significantly increase the processing time, even when all the keys are equal.
- An optimal order preserving hashing function for different key value distributions can be found.
- The initial order of the keys plays a significant role in existing sorting algorithms, but it is not relevant to the performance of the **OTHER** sorting algorithm. However, performance of the LIS sort, which is used to sort collisions, is influenced by the initial order among the keys.
- As expected, the acceleration ratio grows as the cardinality of the database grows because of faster processing per key and diminishing influence of fixed table processing overhead in the case of the **OTHER** algorithm. It can be seen in Fig. 5 that theoretic acceleration increases from 10 ($N = 50$) to 45 ($N = 100,000$).

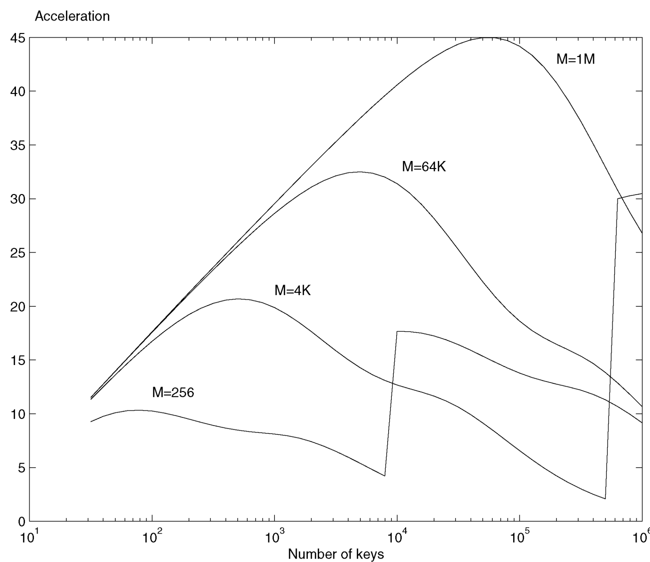

Fig. 5. Acceleration of the Quicksort for varying table size.

4.4 Join

As in traditional database management systems, one can perform either a hash-based join or a sort-merge join.

Accelerated sort and hash operations accelerate the join operation of the **OTHER** algorithms. The range of acceleration is similar to the performance of sort operations.

5 Architecture of the Proposed Accelerator

Our analysis showed that the table processing operations (e.g., *table initialization* and *table scan*) in the **OTHER** algorithms consumes significant amounts of processing time. For example, in the case of sorting, 40 to 95 percent of the processing time is due to the table processing operations depending on the cardinality of the underlying databases. As a result, we introduced the bit vector and bit vector operations and attempted to accelerate these operations. In order to accelerate the aforementioned primitives in a cost-efficient manner, we developed an accelerator that can be used either as an extension to the CPU or as a coprocessor.

5.1 Bit Manipulation Accelerator (BMA)

The **BMA** is used in conjunction with conventional random access memory; together, they simulate an associative memory. The proposed accelerator is designed to accelerate three primitives, *TableInit*, *Mark*, and *GetNext*, as discussed in Section 2.

As noted earlier, sparse bit vectors introduce overhead; as a result, we introduced a hierarchical tree organization for the bit vector. The hierarchical structure of the bit vector offers a fast ***Scan*** and a constant number of memory accesses per scan cycle; however, these advantages come at the expense of increased hardware complexity of the algorithm and a slower ***Mark*** primitive. Consequently, we developed two types of accelerators: The **F**lat **A**ccelerator (**FA**) and the **H**ierarchical **A**ccelerator (**HA**). Nevertheless, regardless of the implementation, the CPU communicates with the accelerator via the following set of registers, either as a set of I/O or as memory mapped ports:

- *ControlRegister* (*CR*) is used to initialize the accelerator to the specific mode of operation (Init, Mark, Scan) and to adjust the size of the bit vector.
- *BitSetRegister* (*BSR*) accepts the bit address as the argument of the ***Mark*** primitive.
- *Bit Test Register* (*BTR*) is a one-bit read-only register, containing the previous status of the cell which is marked during the ***Mark*** primitive.
- *Scan Register* (*SR*) is a read-only register containing the result of the ***Scan*** primitive (i.e., the address of the next marked cell).

The Flat Accelerator (FA): Fig. 6a shows the block diagram of the FA scheme. As mentioned before, a bit in the bit vector is referred by the Word Address (WA) and the Bit Address (BA) within the word. Bit vector scan is performed sequentially; consequently, the address generator is realized as a counter. The *Word Scan Register* (*WSR*) generates the address of the marked bit within the selected word (BA) during the scan phase. The *Scan Register SR* then concatenates *WA* and *BA*, generating a unique logical address. The FA will not take over the system bus if there are more marked bits in the current index word.

In contrast to the CISC implementation of bit operations, the *WSR* generates the next *BA* in the current word during a single cycle. Moreover, the *WSR* automatically clears the marked bit in the current word to enable further scanning within the word. The FA can scan a range of bit addresses; this feature allows us to select the range of key values or a set of hashed values.

The Hierarchical Accelerator (HA): In addition to the basic functionality of the FA, the HA incorporates a more complicated control logic, address generator, and the scanner to allow hierarchical bit vector operations. The block diagram of the HA is given in Fig. 6b. The scanner contains four index registers for current words in every level of hierarchy. The HA also contains four *WSR* registers (one for every level of the hierarchy) to allow rapid scanning through the hierarchical tree. For a 1Mbit vector and a word length of 32-bits, this functionality allows us to access a marked bit in fewer than six memory cycles.

Both accelerators were designed and simulated using the TANNER standard cell VLSI package. The complexity of the FA is 1,800 cells and the HA requires 4,300 cells in the case of 32-bit accelerator [13].

6 Simulation and Comparative Analysis

Simulation results of software implementation of **OTHER** algorithms are given in Table 3. The execution traces are collected on the DEC Alpha 500au workstation with the DEC Alpha 21164/500MHz processor and Unix 4.0d operating system. The performance of the proposed algorithm is compared with the optimized Quicksort algorithm tuned for execution on the target system [29]. Both algorithms sort pointers to the records rather than physically moving the records. The performance of the algorithm using a table of size $M = 64\text{K}$ and $M = 1\text{M}$ entries is reported. As can be concluded, **OTHER** algorithms achieve significant performance improvement, even if implemented in the software. Moreover, table processing consumes 50-95 percent of the processing time in the proposed algorithms. Hence, efficient hardware support for table manipulation operations should significantly improve the performance of the proposed algorithms.

We developed a simulator to evaluate the overall performance of the proposed accelerator based on the operation mix of typical database applications, as reported in the literature. A MIPS-based superscalar CPU with two instructions per cycle is used as the underlying platform [30]. The accelerator could be implemented as the on-chip accelerator tightly coupled to the CPU or an intelligent off-chip bus master. We simulated the on-chip accelerator using special read/write instructions for communication with the accelerator; all instructions that read the result register of the accelerator will be blocked until the results are available. In addition, all subsequent instructions are also blocked. We believe that out-of-order execution, whenever possible, will further increase the performance of **OTHER** algorithms, allowing useful processing while the CPU is waiting for the result of the accelerator. Execution capability will further increase the performance of **OTHER** algorithms, allowing useful processing while the CPU is waiting for the result from the accelerator.

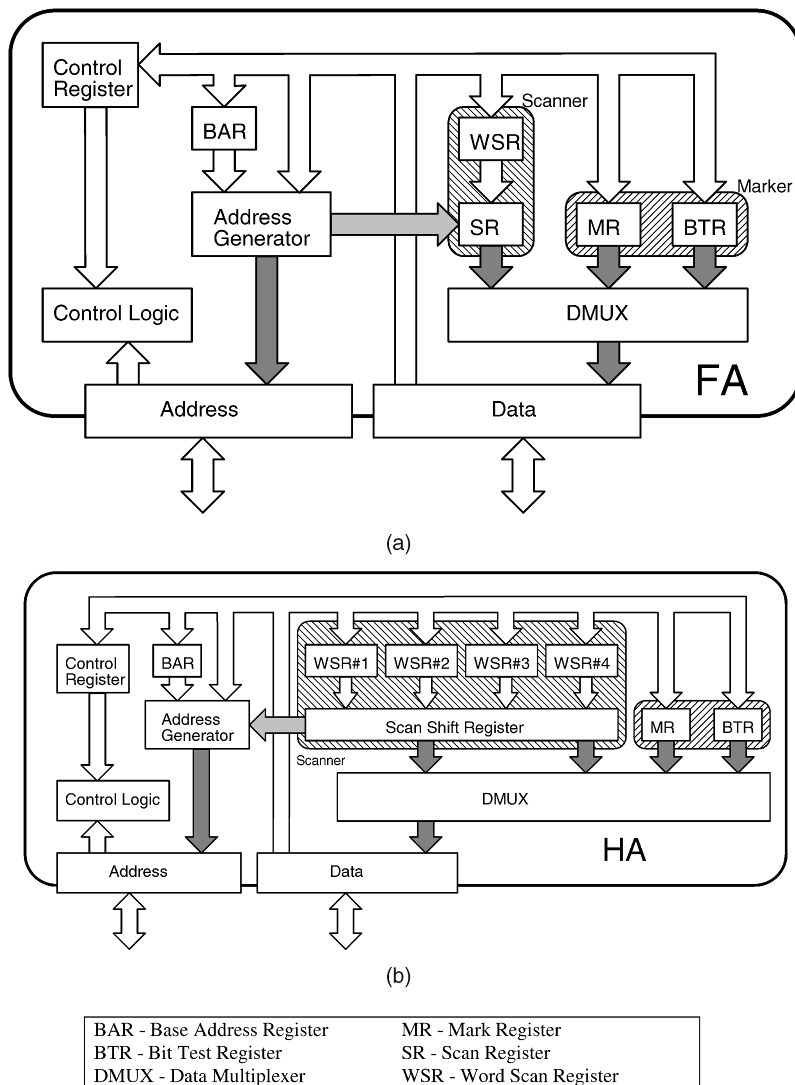

Fig. 6. Block diagram of the proposed accelerator. (a) Flat organization. (b) Hierarchical organization (HA).

The simulator was also extended to compare and contrast the effectiveness of the proposed accelerator against an ideal High-Level Accelerator (**HLA**). We assumed that the HLA had infinite processing speed and internal memory. The processing time would be just a processor time to write the set of keys and read the result as a list of processed identifiers (*tid*). Theoretically, the maximum acceleration of the proposed table hashing algorithms was presented as "**Z**ero time" **T**able **P**rocessing (**ZTP**) performance. It was the performance of the accelerator with infinite speed and private memory which does not require additional system bus cycles.

As anticipated, we found that the performance of the accelerated system is somewhere between the software and ideal implementation of table operations (e.g., **ZTP**)—Fig. 7. The performance depends on table loading factor (*N/M*). If the optimal table size is chosen, stable performance improvement is achieved, as presented in Fig. 8. Finally, as noted before, the **HA** organization offers a better

TABLE 3
Acceleration and Table Processing Overhead of **OTHER** Sorting Algorithm

Number of keys	**OTHER** M=64K		**OTHER** M=1M		Cycles/data		
	Acceleration	Table processing	Acceleration	Table processing	Quicksort	**OTHER** M=64K	**OTHER** M=1M
10,000	3.402	82.49%	1.286	94.45%	988	290	768
100,000	6.439	51.86%	3.606	86.13%	1245	193	345
1,000,000	3.190	47.45%	7.955	58.73%	1557	488	196

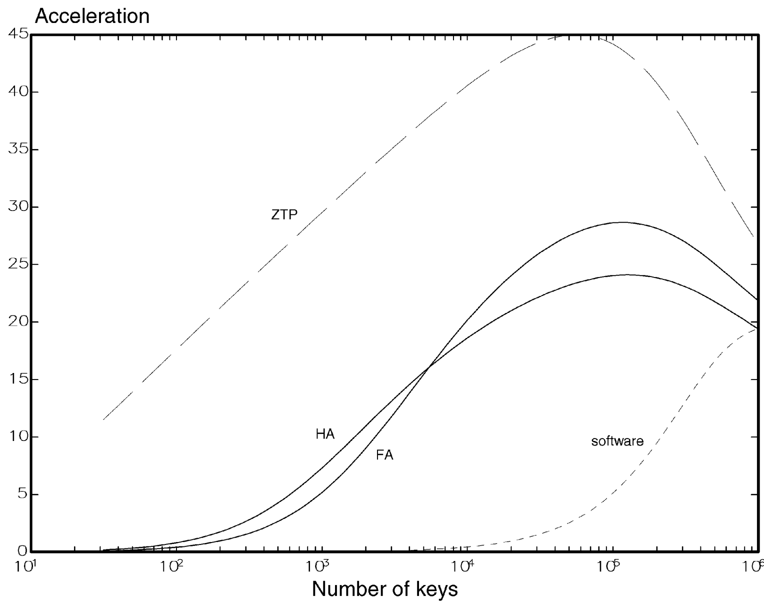

Fig. 7. Comparative analysis of the proposed accelerator, ideal accelerator, and a software approach.

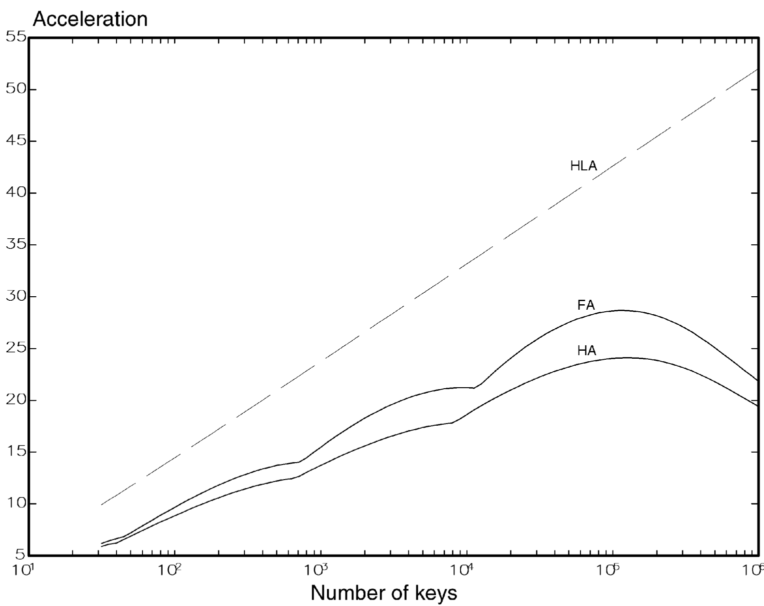

Fig. 8. Acceleration of sort operation.

performance when the hash tables are sparse (small table loading factor).

The simulated cache performance of the **OTHER** sorting algorithm is given in Table 4. The execution traces were collected on the DEC Alpha 500au workstation with the DEC Alpha 21164/500MHz processor [31] and analyzed using the ATOM cache analysis tool [32]. The simulation conditions were the same as explained in Section 2. The

TABLE 4
Cache Performance of the **OTHER** Sorting Algorithm

Number of keys	Quicksort		**OTHER** TableSize=64K			**OTHER** TableSize=1M		
	Data ref/key	Cache miss ratio	Data ref/key	Cache miss ratio	Data ref. ratio	Data ref/key	Cache miss ratio	Data ref. Ratio
10,000	352	3.65%	88	3.64%	3.96	202	2.07%	1.74
100,000	440	4.39%	70	5.69%	6.22	101	3.69%	4.35
1,000,000	547	4.71%	122	4.09%	4.48	70	5.97%	7.82

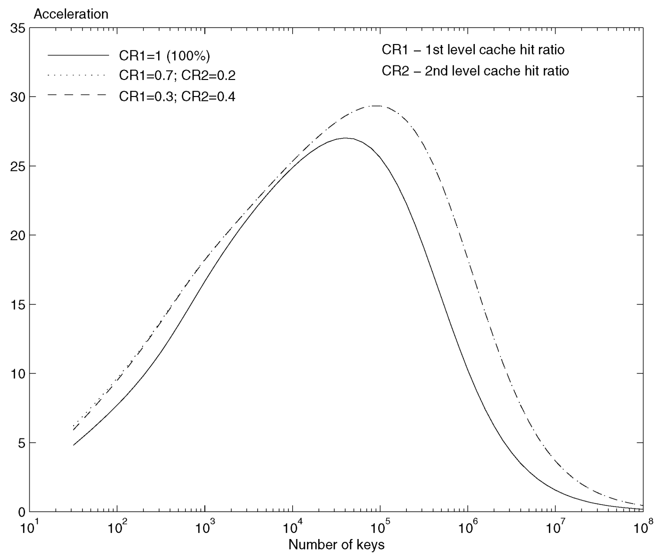

Fig. 9. FA cache performance for sorting with single pass table preprocessing.

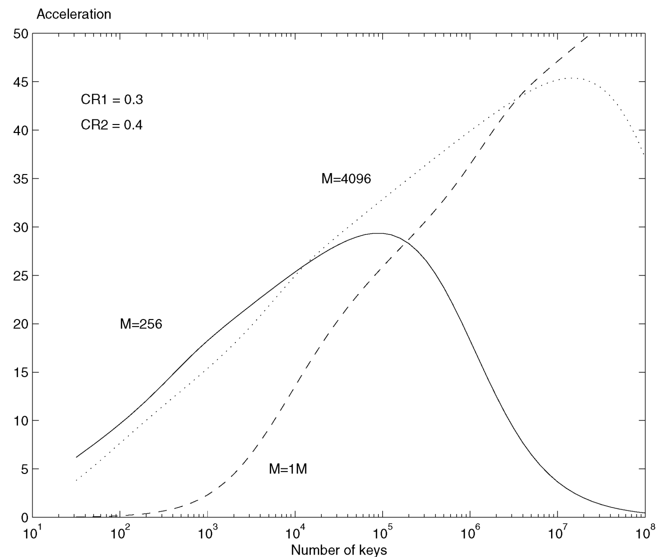

Fig. 10. Optimal FA performance with variable table size and data cache hit ratio 70 percent.

data cache was simulated using an 8KB cache with direct mapping and 32 byte blocks. It is clear that the **OTHER** algorithm has a significantly lower number of data references per sorted key than the Quicksort algorithm. The best software is achieved when table size is between $N/2$ and N, where N is the number of processed keys. The data cache miss ratio is similar to that for Quicksort, even with 4 to 8 times fewer data references.

The effect of cache memory on the performance of the proposed accelerator architecture model was also simulated. We assumed a separate two-level instruction and data caches, with access times of one and four processor cycles, respectively. Our simulator also assumed a main memory access time of 16 processor cycles. Finally, it was assumed that all instructions are fetched from the first-level cache. This assumption was mainly due to the relatively small size and repetitive nature of both Quicksort and the proposed **OTHER** algorithm. Fig. 9 depicts the effect of the cache on the performance of the FA accelerator with single-partition preprocessing for the sort operation based on different cache efficiency (table size M = 256). The relative performance improvement for lower cache efficiency could be explained by the lower number of data references per sorted key and the higher locality of data access (Table 4). However, such an advantage diminishes when large tables are used to process a small number of keys. Since **OTHER** algorithms use optimal table size, cache efficiency further increases the relative performance advantage of the proposed solution (Fig. 10).

We also examined the overall performance improvement of database operations. Our testbed included *select*, *sort*, and some typical database queries containing a mixture of database operations [33], [34]. The choice of the ideal HLA was due to the fact that we intended to show that our scheme achieves a similar performance improvement at the expense of fewer hardware resources. This can be contributed to the generality and effectiveness of the **OTHER** primitive operations. Fig. 7 shows the acceleration of *sort* operations. Significant acceleration of *select* operations could be achieved only for the ideal HLA (up to 80 percent). The proposed accelerators do not achieve significant performance improvement due to the simplicity and regularity of the *select* operation in the software implementation (FA accelerates the *select* operation just for 8 percent when $N > 2,000$).

Fig. 11 presents the overall acceleration of database queries for a typical mixture of database operations [33], [35]. It should be noted that our simulation analysis also shows that, for a select on multiple attributes, the **OTHER** accelerator offers superior performance over the **HLA**.

7 Conclusion

Efficient handling of large databases motivated our research. We intended to develop a simple and cost efficient accelerator for a set of primitive and general nonnumeric

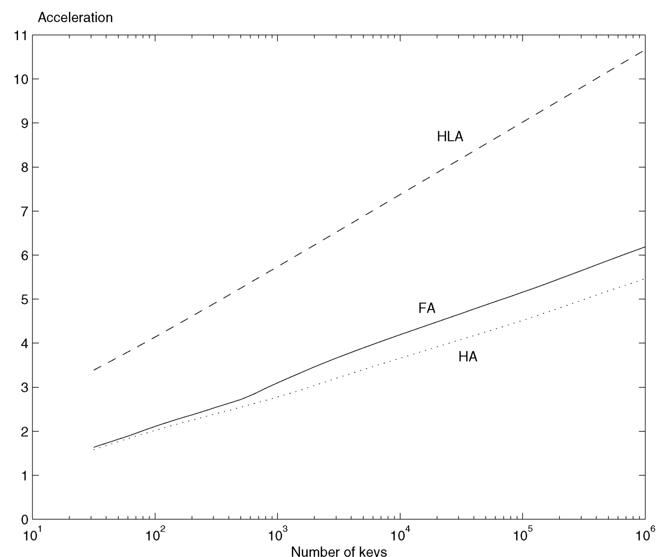

Fig. 11. Acceleration of database queries.

operations. As a result, the so-called **OTHER** primitive operations were introduced. In addition, we demonstrated how complex database functions could be mapped into the **OTHER** primitives. Analytical and simulation studies were reported to discuss the effectiveness of our approach. We have shown that a comparable performance to an ideal HLA can be achieved by using a very cost efficient accelerator. Finally, different designs for our accelerator were introduced and analyzed.

The proposed algorithms are very efficient, even in software implementation. We have found that the optimal table size for software implementation on the DEC Alpha 500au is between $N/2$ and N, where N is the number of processed keys. The proposed accelerators make efficient use of small hash tables (Fig. 9), at the expense of additional memory area for S and C vectors.

It is surprising to find that our flat accelerator (FA), although requiring three times fewer gates to implement than the hierarchical accelerator (HA), in almost all cases demonstrated a higher performance. This suggests that we can use smaller tables and, hence, partition the keys into smaller units, along with incorporating the FA accelerator design.

REFERENCES

[1] L.L. Miller, A.R. Hurson, and S.H. Pakzad, *Parallel Architectures for Data/Knowledge Based Systems,* IEEE Tutorial. IEEE CS Press, 1995.
[2] C.V. Ramamoorthy, J.L. Turner, and B.J. Wah, "A Design of a Fast Cellular Associative Memory for Ordered Retrieval," *IEEE Trans. Computers,* vol. 27, no. 9, pp. 800-815, Sept. 1978.
[3] C. Lee, S.Y. Su, and H. Lam, "Algorithms for Sorting and Sort-Based Database Operations Using a Special-Function Unit," *Database Machines and Knowledge Base Machines,* M. Kitsuregawa and H. Tanaka, eds., pp. 103-116, Kluwer Academic, 1988.
[4] K.C. Lee, T.M. Hickey, V.W. Mak, and G.E. Herman, "VLSI Accelerators for Large Database Systems," *IEEE Micro,* pp. 8-20, Dec. 1991.
[5] E. Jovanov, D. Starcevic, T. Aleksic, and Z. Stojkov, "Hardware Implementation of Some DBMS functions Using SPR," *Proc. 25th Hawaii Int'l Conf. System Sciences,* vol. 1, pp. 328-337, Jan. 1992.
[6] M. Kitsuregawa, S. Hirano, M. Harada, M. Nakamura, and M. Takagi, "The Super Database Computer (SDC): System Architecture, Algorithm, and Preliminary Evaluation," *Proc. 25th Hawaii Int'l Conf. System Sciences,* vol. 1, pp. 308-319, Jan. 1992.
[7] B. Parhami and D.M. Kwai, "Data Driven Control Scheme for Linear Arrays: Application to a Stable Insertion Sorter," *IEEE Trans. Parallel and Distributed Systems,* vol. 10, no. 1, pp. 23-28, Jan. 1999.
[8] E. Babb, "Implementing a Relational Database by Means of Specialized Hardware," *ACM Trans. Database Systems,* vol. 4, no. 1, pp. 1-29, Mar. 1979.
[9] U. Inoue, T. Satoh, H. Hayami, H. Takeda, T. Nakamura, and H. Fukuoka, "RINDA: A Relational Database Processor with Hardware Specialized for Searching and Sorting," *IEEE Micro,* pp. 61-70, Dec. 1991.
[10] M. Kitsuregawa, W. Yang, T. Suzuki, and M. Takagi, "Design and Implementation of High Speed Pipeline Merge Sorter with Run Length Tuning Mechanism," *Database Machines and Knowledge Base Machines,* M. Kitsuregawa and H. Tanaka, eds., pp. 89-102, Kluwer Academic, 1987.
[11] D.H. Andrews et al., "The AS/400 Alternative," ADM Inc., 1990.
[12] *80386 Programmer's Reference Manual.* Santa Clara, Calif.: Intel, 1986.
[13] E. Jovanov, "The Architecture of Accelerator for Database Operations," PhD thesis, School of Electrical Eng., Univ. of Belgrade, Yugoslavia, 1993.
[14] R.D. Sloan, "A Practical Implementation of the Database Machine —Teradata DBC/1012," *Proc. 25th Hawaii Int'l Conf. System Sciences,* vol. 1, pp. 320-327, 1992.
[15] W.D. Maurer and T.G. Lewis, "Hash Table Methods," *ACM Computing Surveys,* vol. 7, no. 1, pp. 5-19, Mar. 1975.
[16] E.J. Isaac and R.C. Singleton, "Sorting by Address Calculation," *J. ACM,* no. 3, pp. 169-174, July 1956.
[17] M.D. MacLaren, "Internal Sorting by Radix Plus Sifting," *J. ACM,* vol. 13, no. 3, pp. 404-411, July 1966.
[18] T. Aleksic, Internal Reports DBC/1-DBC/15, School of Electrical Eng., Univ. of Belgrade, Yugoslavia, 1986.
[19] E. Jovanov, T. Aleksic, Z. Stojkov, and D. Starcevic, "A Sorting Processor for Microcomputers," *Microprocessing and Microprogramming,* vol. 23, nos. 1-5, pp. 273-278, 1988.
[20] Z. Stojkov, "An Implementation of Relational Algebra Operations," MS thesis, School of Electrical Eng., Univ. of Belgrade, Yugoslavia, 1993.
[21] M.V. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient Hardware Hashing Functions for High Performance Computers," *IEEE Trans. Computers,* vol. 46, no. 12, pp. 1378-1381, Dec. 1997.
[22] D.E. Knuth, *The Art of Computer Programming, Vol. 3, Sorting and Searching.* Reading, Mass.: Addison-Wesley, 1973.
[23] W. Feller, *An Introduction to Probability Theory and Its Applications,* vol. 1. New York: John Wiley & Sons, 1968.
[24] L. Raschid, T. Fei, H. Lam, and Y.W. Stanley, "A Special Function Unit for Sorting and Sort-Based Database Operations," *IEEE Trans. Computers,* vol. 35, no. 12, pp. 1071-1077, Dec. 1986.
[25] C.J. Date, *An Introduction to Database Systems,* seventh ed. Reading, Mass.: Addison-Wesley, 2000.
[26] C.J. Date and H. Darwen, *A Guide to SQL Standard.* Reading, Mass.: Addison-Wesley, 1993.
[27] B.H. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Comm. ACM,* vol. 13, no. 7, pp. 422-426, July 1970.
[28] J.K. Mullin, "Optimal Semijoins for Distributed Database Systems," *IEEE Trans. Software Eng.,* vol. 16, no. 5, pp. 558-560, May 1990.
[29] J.L. Bentley and M.D. McIlroy, "Engineering a Sort Function," *Software Practice and Experience,* vol. 23, no. 11, pp. 1249-1265, Nov. 1993.
[30] J. Henessy, N. Jouppi, S. Przybylski, C. Rowen, and T. Gross, "Design of a High Performance VLSI Processor," Technical Report no. 236, Stanford Univ., Palo Alto, Calif., Feb. 1983.
[31] http://www.ece.uah.edu/~jovan/equipment/Alpha au-series.htm, 2002.
[32] A. Eustace and A. Srivastava, "ATOM: A Flexible Interface for Building High Performance Program Analysis Tools," DEC WRL Technical Note TN-44, July 1994.
[33] H. Boral and D.J. DeWitt, "A Methodology for Database Performance Evaluation," *Proc. Fourth Int'l Workshop Database Machines,* pp. 166-187, 1984.
[34] D.J. DeWitt, S. Ghandeharizadeh, D. Schneider, R. Jauhari, M. Muralikrishna, and A. Sharma, "A Single User Evaluation of the Gamma Database Machine," *Database Machines and Knowledge Base Machines,* M. Kitsuregawa and H. Tanaka, eds., pp. 370-386, Kluwer Academic, 1987.
[35] P.S. Yu, M.S. Chen, H.U. Heiss, and S. Lee, "On Workload Characterization of Relational Database Environments," *IEEE Trans. Software Eng.,* vol. 18, no. 4, pp. 347-355, Apr. 1992.

Emil Jovanov received the Dipl Ing (1984), MSc (1988), and PhD (1993) degrees in electrical engineering from the University of Belgrade, Yugoslavia. From 1984-1998, he worked at the Research Institute "Mihajlo Pupin" in Belgrade. From 1994 to 1998, he was an assistant professor at the School of Electrical Engineering, University of Belgrade. He is currently an associate professor in the Electrical and Computer Engineering Department at the University of Alabama in Huntsville. His research interests include database systems, computer architecture, parallel and distributed systems, biomedical signal processing, and ubiquitous computing, in which he has published more than 100 papers. He is a member of the IEEE and ACM.

Veljko Milutinovic (M'81-SM'85) received the PhD degree from the University of Belgrade, Serbia, Yugoslavia, in 1982. He has been with the Department of Computer Engineering, School of Electrical Engineering, University of Belgrade since 1990. Prior to that, he was on the faculty of Purdue University, West Lafayette, Indiana. His research interests are in computer architecture/design, as well as in system support for electronic business on the Internet. He has contributed more than 50 papers to IEEE journals on computer architecture/design and technology-aware system support for mission-critical applications. He has consulted for leading industries in the US and Europe (IBM, RCA, NCR, AT&T, Virtual, eT, Zycad, Aerospace Corporation, Electrospace Corporation, Intel, Fairchild, Honeywell, Encore, Phillips, etc.). He was involved in a number of market successful industrial efforts (designer of the first multi-processor HF data modem in the 1970s, coarchitect of the first 200MHz RISC microprocessor in the 1980s, project leader of the first RMS system for personal computers in the 1990s). He is the author of several books and was an editor/coeditor for a number of IEEE tutorial books and conference proceedings. Dr. Milutinovic has served as a guest editor for special issues of the *Proceedings of the IEEE*, *IEEE Transactions on Computers*, *Computer*, and *IEEE Concurrency*. He has presented more than 300 invited talks around the world. He is a senior member of the IEEE.

Ali R. Hurson is a member of the Computer Science and Engineering Faculty at The Pennsylvania State University. His research for the past 18 years has been directed toward the design and analysis of general as well as special purpose computer architectures. His research has been supported by the US National Science Foundation, Office of Naval Research, Defense Advanced Research Projects Agency, NCR Corp., IBM, Lockheed Martin, and Penn State University. He has published more than 200 technical papers in areas including database systems, multidatabases, object-oriented databases, computer architecture and cache memory, parallel and distributed processing, dataflow architectures, and VLSI algorithms. Dr. Hurson served as the guest coeditor of special issues of the *IEEE Proceedings* on supercomputing technology, the *Journal of Parallel and Distributed Computing* on load balancing and scheduling, the *Journal of Integrated Computer-Aided Engineering* on multidatabase and interoperable systems, *IEEE Transactions on Computers* on parallel architectures and compilation techniques, and the *Journal of Multimedia Tools and Applications*. He is the coauthor of the *IEEE Tutorials on Parallel Architectures for Database Systems, Multidatabase Systems: An Advanced Solution for Global Information Sharing, Parallel Architectures for Data/Knowledge Base Systems, and Scheduling and Load Balancing in Parallel and Distributed Systems*. He is also the cofounder of the IEEE Symposium on Parallel and Distributed Processing (currently IPDPS). Professor Hurson has been active in various IEEE/ACM conferences and has given tutorials for various conferences on global information sharing, dataflow processing, database management systems, supercomputer technology, data/knowledge-based systems, scheduling and load balancing, and parallel computing. He served as a member of the IEEE Computer Society Press Editorial Board, an editor of the *IEEE Transactions on Computers*, and an IEEE Distinguished Speaker. Currently, he is serving on the IEEE/ACM Computer Sciences Accreditation Board and as an ACM lecturer. He is a senior member of the IEEE.

▷ **For more information on this or any computing topic, please visit our Digital Library at** http://computer.org/publications/dlib.