Extended ASLM - A Reconfigurable Database Machine

A.R. Hurson *, L.L. Miller **, S.H. Pakzad *, and D.B. Shin *

* Computer Engineering Program
The Pennsylvania State University
University Park, PA 16802

** Department of Computer Science
Iowa State University
Ames, Iowa 50011

Abstract

Society is experiencing an information explosion. The proliferation of data combined with higher performance requirements have forced the designers of database systems to look beyond software solutions. Since the early seventies, hardware approaches, namely database machines, have been introduced to overcome the inefficiencies of the software database systems. Most of the database machine designs have been based on the relational model, in addition, the performance of these machines has been closely identified with the performance of the join operation.

The present work studies the architectural features of the ASLM database machine and examines the retrieval aspect of incorporating a comprehensive null value policy into the design of ASLM. The join operation is used as a performance measure in the evaluation of ASLM. In addition, the extension of ASLM to a multiuser/multiprogram environment via dynamic reconfigurability of the hardware resources among a set of concurrent queries is discussed.

1. Introduction

As database systems have become more sophisticated, users have increasingly looked to the system designers for additional support in processing their applications. Moreover, the proliferation and distribution of data, higher performance requirements, and the issue of the fault tolerance capability of the database systems have forced designers of such systems to look beyond traditional software solutions in a centralized environment.

A database machine is defined as a dedicated system capable of performing all or part of database management functions in hardware, software or a combination of both. Such a radical departure from the conventional approach in handling databases is due to the existing semantic gap between database systems and conventional architectures. Several different classifications of database machines have been proposed in the literature. Parameters such as: the functionality of a database machine in computing environment [1-5], the concept of parallelism, and the location where the hardware logic is applied [6-8] have been used as the basis of these classifications. However, one common feature of such classifications is the existence of a special purpose hardware system designed to manipulate the database. This approach is based on the need to remove the database functions from the host machine. The hardware specification of the database operations in the backend computer, the ability to overlap operations between the host and the backend, and elimination of the problems associated with a 90-10% rule [4] have made the backend approach more promising than the other categories.

The majority of the database architectures which have been advanced in the literature are based on the relational model [12-13]. This is due to the: i) Similarity between representation of data in the relational model and user's conceptual view of the data files, and ii) The mathematical foundation on which the relational data model is based. In practice, the performance of a relational database system is tied to the performance of the join operation. Such an association is due to the complexity of the join and its practicality in the database queries. This complexity directly affects the time and space efficiency of the architecture, and hence the overall performance of the system. The ability of our model in handling the join operation will be used as a measure to analyze its performance.

The objective of this paper is three fold: First, it discusses the overall architecture of the Associative Search Language Machine (ASLM) and its extension to a multiplayer MIMD (Multiple Instruction stream, Multiple Data stream) architecture. Secondly, it examines the maybe algebra operators and its incorporation in the ASLM design. Finally, the performance of the extended ASLM with respect to the join operation using the discrete event simulation technique is analyzed.

2. ASLM - A Backend Database Machine

ASLM (Associative Search Language Machine) is a backend database machine [14]. The system is composed of a general purpose frontend machine supported by ASLM (Figure 1). The frontend system acts as the interface between the user and the backend machine. Security validation of the user and the user's query [15], translation of the user's query into ASL primitives [16], and transmission of the final results to the user are the major functions of the frontend system.

Reducing the semantic gap, alleviating the data communication, and designing an architecture which satisfies the technological constraints have been the general motivation behind the design of ASLM. Semantic gap reduction has been achieved through: i)
the generation of microcode for ASLM micro sequences, ii) the one to one relationship between the set operations and associative operations, iii) hardware implementation of the basic relational operations and iv) elimination of the address accessibility of the data. The data communication problem has been resolved by screening the data close to the secondary storage. Finally, ASLM is highly modular and each module is composed of a few basic and simple units replicated in a two dimensional space. In addition, in contrast to some other database machines, ASLM is not dependent on any specific secondary storage technology. Solving these problems has determined the organization of ASLM.

ASLM is composed of four modules: i) controller, ii) secondary storage interface, iii) an array of preprocessors, and iv) a database processor.

Controller: The controller is a micro-programmable control unit. The contents of the writable control memory will be determined by the user's program. To combat the growth in computer crime, and the increase in database complexity and size, security mechanism is enforced during the compilation. Two classes of security violations namely name dependent and content/context dependent access control are distinguished by the security processor. The name dependent access control violations are detected and enforced during the compilation. However, the content/context dependent access controls are detected during the compilation, but enforced during the execution phase. This is done by modifying the user's query during the compilation according to his/her security capability. More specifically, the controller: i) stores the ASL microinstructions generated by the ASL compiler, ii) decodes the ASL microinstructions, and iii) propagates the control sequences to the appropriate modules in the backend. The simplicity of the controller is primarily due to the inherent parallelism and content addressability of the associative operations.

Secondary Storage Interface: The secondary storage interface is a collection of random access modules, augmented by some hardware facilities. This module can be assumed as a shared I/O cache between the ASLM and the secondary storage units. This allows ASLM to utilize recent and future advances in storage technology without any architectural effect on the design. Blocks of data are prefetched into the cache modules and then distributed among the preprocessors. Due to the facts that ASLM is based on the use of unordered data files, and the storage operations (e.g. insert, delete and modify) are not performed in place, one can develop very simple lookahead and replacement algorithms to govern the flow of data and control in the proposed I/O cache module.

Preprocessors: Because of the practical limitation on the size of the associative modules incorporated in the database processor, it may not be possible to store all available data in the associative memory in order to perform an operation on the data items. Due to this restriction, all the systems proposed earlier, based on fully associative memory, are not capable of handling large databases [2,4,5]. In contrast, the design of ASLM overcomes this problem through consideration of two points. First, queries almost always refer to a subset of attributes in the tuples. Second, in each query, a small subset of tuples will satisfy the search criteria [4].

The preprocessors act as a filter which screens the data, selecting the valid data. The valid data is then placed in an associative module in the database processor. In addition, the preprocessors perform a projection over the relevant attributes. In a database operation, attributes of a relation can be classified into three groups: i) members of the search argument (SA), ii) members of the output set (OS), and iii) the remaining attributes. It is the task of the preprocessors to validate tuples, on the basis of the SA attributes and to project them over the OS attributes.
As the associative modules can be linked together to make a memory of size (k*w)*d, the associative modules act as intermediate storage which holds part of the relation in order to perform the required relational operations on it. The database processor receives the database operations from the controller and performs them on the data stored in the appropriate associative modules. At the end of the operations defined by the query, the result is transferred to the general purpose computer.

The database processor is a set of identical and independent associative modules of size w*d (where w is the width and d is the depth) augmented by some hardware circuits. The associative modules can be linked together to make a memory of size (k*w)*d (1 ≤ k ≤ n, where n is the number of associative modules) capable of holding d tuples of size k*w bits, or they could be linked to form a memory of size w*(k*d) capable of holding k*d tuples of size w bits. Because of the variability of the tuple sizes in a database, this facility enables the system to adjust the available memory modules based on the length of the tuples and the cardinality of the relation. In addition, this increases the space efficiency of the database processor. The independence of the associative modules enhances the modularity and as such the fault tolerance of the system.

To reformat and realign tuples during some operations such as join (i.e. organizing join attributes(s) in the source tuples according to the position of join attributes(s) in the target tuples), the database processor is enhanced by a small special purpose unit, called the concatenation unit. This unit is capable of performing basic logic operations on a collection of shift registers.

ASLM is based on the concept of variable length tuples, where tuples and attribute fields are separated from each other by tuple separator markers and attribute separator markers, respectively. The null value is represented in this format as two adjacent attribute separator markers. The fields of a tuple t defined by t(OS) are expanded by the preprocessors to their predefined size in the normal manner. Fields containing null values are expanded to their appropriate size with don't care symbols.

3. Maybe Algebra

Missing information presents a problem for any data model and as the relational model has matured, researchers have examined the questions of how to handle missing data. Codd [12] developed an extension to relational algebra that allows a user to retrieve information based on partial results.

Relational operators using a search condition to select tuples will base the search on a set of attributes, say X, that fall within a single relation scheme. Clearly the selection of a tuple (t) based on a given condition will depend on the state of the values for t(X). The truth value maybe (ω) is assigned if ∀ A ∈ X such that t(A) is definite on A, the value t(A) matches the search condition and there exists at least one element of X, say B, such that t(B) = ω. Clearly, such an approach has the potential to return nonrelevant tuples to the user. The tuple matching process of the maybe join of r and s (written r(R) m ASLM s(S)) can be defined on the basis of a boolean expression E. For a join over d attributes, we have E=C1 ∧ C2 ∧ ⋯ ∧ Cd where Cj = A θ B, A, B are θ compatible, and A ∈ r, B ∈ s. For the natural join over the join attributes A1, A2, …, Ad, we have E=(r.A1 = s.A1) ∧ (r.A2 = s.A2) ∧ ⋯ ∧ (r.Ad = s.Ad). A tuple tr ∈ r is joined with a tuple ts ∈ s if E evaluates to maybe for the two tuples. The reason for the possibility of returning nonrelevant tuples is that the operator does not incorporate user knowledge of restrictions on the individual Cj 1≤j≤d into the join decision. Rather, the decision to join tr and ts is only based on the final value of E.

In [17-18] we proposed a new set of operators to reduce the number of nonrelevant tuples returned. The new operators are based on the observation that the user generally will have enough information about the desired result to predetermine that some of the Cj 1≤j≤d should evaluate to true while E evaluates to maybe (ω). We call the new operators attribute maybe operators to distinguish them from Codd's maybe operators. Briefly, they are:

i) attribute maybe join
Let r(R) and s(S) be two relations. Let E be the join condition in conjunctive normal form (CNF). Let E = C1 ∧ C2 ∧ ⋯ ∧ Cm where Cj 1≤j≤m is either AθB (evaluates to true) or AθB (evaluates either true or maybe) where A ∈ r and B ∈ s. A tuple tr is returned by the attribute maybe join on r and s (written r [\text{ ASLM }] s) if E evaluates to maybe for tr and ts.

ii) attribute maybe select
Let r(R) be a relation. Let E be a selection condition defined over attributes in R. E contains two types of comparison operators θ (evaluates to true) and (evaluates to either true or maybe). A tuple ts is selected for the attribute maybe select (written s [\text{ ASLM }] (r)) if E evaluates to maybe for ts.
iii) attribute maybe divide

Let \( R = (R_1, R_2, ..., R_k) \) and \( S = (S_1, S_2, ..., S_k) \).

Let \( r(R) \) and \( s(S) \) be relations such that, for each \( S_i \times S_j \) there exists a \( \theta \) compatible \( R_i \times R_j \) such that \( E = C_1 \land C_2 \land ... \land C_k \) be a condition in CNF such that \( C_1 \) is either \( S_i \ominus R_j \) or \( S_j \ominus R_i \) with the aforementioned interpretation. A tuple is returned by attribute maybe divide, if the division operation evaluates to maybe under \( E \).

As shown in the example below, the attribute maybe join operation provides superior logical performance when the restrictions apply. In addition, performance as well when the restrictions can be viewed as retrieving tuples with unknown critical values. It is shown in a later section that it has superior performance to maybe join, and division. A somewhat different view of these operations is given above in the relational algebra. To accomodate large relations, ASLM stages the relation between auxiliary memory and the associative module.

4. Performance Evaluation of the Database Processor

Because of the complexity of the join and its practical application in a user's query, the performance of DBMS based on the relational model has been tied to the performance of the join operation [19-20]. ASLM performs a parallel associative version of the nested loop algorithm [21].

In the remainder of this section, we examine the action of the join module for the True Join, Maybe Join and Attribute Maybe Join operations. We start the discussion by looking at some features of the operations that are common to all three types of join.

In ASLM, the two relations to be joined are loaded into individual associative modules: the source module (\( S_1 \)) and the target module (\( S_2 \)). The tuples in \( S_1 \) are tested one at a time against the tuples in \( S_2 \). Tuples in \( S_2 \) that satisfy the join condition for a source relation tuple are joined with the source tuple and placed in a third associative module (the destination module - \( S_3 \)).

It is natural to assume that one or more of the relations will be too large to allow its being stored entirely in an associative module. To accomodate large relations, ASLM stages the relation between auxiliary memory and the associative module.

4.1 True Join in ASLM

The true join requires some additional preprocessing of the tuples involved in the join. The problem arises when tuples contain null values in their join fields. For example, in an equi-join, two fields containing null values would return a truth value rather than the maybe value (\( \omega \)). To avoid such an anomaly, ASLM uses a presearch to exclude such tuples from consideration. In case of an equi-join operation, ASLM marks off tuples from consideration in the source relation. However, in case of a \( \theta \) join operation, ASLM marks off nonrelevant tuples in both the source and target relations: i.e.

\[ t \in S_1 \land \exists A \land (t[A]=\omega) \]
\[ t \in S_2 \land \exists A \land (t[A]=\omega) \]

The relational operators that make use of a search condition to manipulate relation(s) are select, join, and division. A somewhat different view of the null value is taken for the remaining relational-algebra operators. For these operations, null values in different tuples are interpreted as being equivalent. As such, there is no change in these operations over their use in traditional relational algebra.

If traditional relational algebra is thought of as true (T) algebra, then Codd's maybe algebra can be viewed as retrieving tuples with unknown critical values. The search conditions for such tuples evaluate to maybe (\( \omega \)). Since the subset of the search condition attributes that is definite matches the search criteria and the remainder are null, it is unclear whether or not such tuples are of use to the user. Maybe algebra gives the user the opportunity to investigate the set of tuples and draw his/her own conclusions.

4.2.1 T join in ASLM

The maybe join operation places a tuple in the result relation when the test for a join between two tuples results in a maybe truth value. For each tuple in \( S_3 \), the \( S_2 \) is searched \( 2^d \) times, where \( d \) is the number of definite attributes in the join set. As in the maybe select, the searches are looking for tuples in \( S_2 \).
with different combinations of the definite attributes of the join set that contain one or more null values.

Maybe join has the potential to have drastic effects on the resource utilization and the performance of a database system. For example, if we wish to use the maybe join to join \( R_1 \) and \( R_2 \) over \( A = B \), where \( R_1 \) and \( R_2 \) are the respective schemes and \( A \in R_1 \) and \( B \in R_2 \), then we get one copy of \( R_2 \) for each tuple \( t_1 \) in \( R_1 \) where \( t_1(A) \) is null. Similarly, we get a copy of \( R_1 \) for each tuple \( t_2 \) in \( R_2 \) where \( t_2(B) \) is null.

Interestingly, the loss in physical performance is paralleled by a loss in logical performance. The value of such an operator must be rated by its ability to supply a "useable" set of tuples which the user can examine to see the potential relationships between data values. The relation sizes suggested by the previous example fall beyond the useable category [17]. Based on this discussion, we present a practical restriction on the maybe join operator. In its current form, the operator will likely cause overflow in ASLM when either the number of attributes in the join set is small or the two relations have a high percentage of null values for the join attributes. To deal with this problem, two solutions are being incorporated into the ASLM design.

- a) Maybe join over a single attribute is detected by the ASL compiler and ignored. The user is notified that such an operation is expected to result in a low information result.
- b) During the maybe join operation on two or more attributes, the system will be able to detect that a large percentage of nulls are resulting in a large number of maybe joined tuples. When the module containing the maybe join result reaches a predefined load density, the user is notified of the low information quality of the data in order to determine the continuation or termination of the operation. By using the relative sizes of the two relations involved in the join, the system can set the predefined load density for the module in question.

The use of these two restrictions will result in higher information value results. The first decision can be made during the compilation while the second decision is determined during the execution time.

4.3 Attribute Maybe Join in ASLM

The attribute maybe join is basically a combination of the true join and the maybe join. At compile time, the compiler creates descriptors defining the role of the join attribute in the operation. The descriptors are used to partition the join attributes into two classes—true join attributes and free join attributes.

The operation is initiated by presearching \( S_g \) over the join attributes required to evaluate to true. If in the true join, the presearch is used to remove the possibility of achieving a true match over null values. Once the presearch has been completed, the source tuple \( t \) is reformatted to form the test tuple \( t_t \). The values of the free join attributes (i.e., not restricted to the true truth value) are used to determine the mask. As in the maybe join, any free join attributes containing a null value are masked out. The resulting mask is moved to the mask register of \( S_T \).

After the mask is in place, the values in \( t_t \) are used to generate the appropriate searches. The number of required searches is \( 2^{d'-1} \), when all the join attributes in \( t_t \) are definite (\( d' \) is the number of join attributes required to have a true match). The value reduces to \( 2^{DEF(t_t)-d} \) when one or more free join attributes contain a null value. Consider the set of join attributes \( \{A, B, C\} \) where \( A \) and \( B \) are restricted to true and \( C \) is the only free attribute. In this case, for the test tuple \( t_t = <a, b, c> \) only the search condition \( a = b \) needs to be tested against \( S_T \).

The following algorithm shows the sequence of the operations for the attribute maybe join. It is clear from the example in Section 3 and the following algorithm that the attribute maybe join has excellent potential to provide superior performance over the maybe join operator.

Algorithm 1: Attribute Maybe Join Operation

For each of the attributes required to have a true match

\begin{verbatim}
for all \( t \in \text{source relation} \) do
    begin
    create the test tuple \( t_t \);
    determine the mask;
    set the mask register of \( S_T \);
    let \( P \) be the set of search conditions from \( t_t \);
    for each condition in \( P \) do
        set the comparand register of \( S_T \) to the search condition;
        associative search \( S_T \); 
        pass the selected tuples one at a time and join with \( t \) and place the result in \( S_D \).
    end.
end.
\end{verbatim}

4.4 Null Values and the Modification Operations

The use of null values in a database system increases the complexity of the insertion policy (i.e., concepts such as tuple subsumption and null replacement as well as traditional concerns like duplicates must be considered). Recent literature has addressed a wide variety of such techniques [9]. Creating an insertion policy from the available techniques requires establishing the assumptions on how the system will be used, our approach is to use somewhat of a compromise of the available techniques based on the following assumptions:

- i) A new tuple can be inserted into an ASLM relation if it is not already subsumed by a current tuple.

- ii) There are no hard violations of the database semantics in the current copy of the database.

The first assumption means that if a tuple such as \( t_1 = \)
(\textlangle A_1, \textlangle A_2, \ldots, A_n, \epsilon \rangle \textrangle \ (\epsilon \text{ means empty attribute}) \text{ currently exists in an ASLM relation with scheme \{A, B, C, D, E\} then the insertion of a new tuple t_2 = \textlangle A_1, \textlangle A_2, \ldots, A_n, A_{n+1}, \epsilon \rangle \text{ means that the tuple t_1 = \textlangle A_1, \textlangle A_2, \ldots, A_n, \epsilon \rangle \textrangle \text{ is no longer needed since t_2 subsumes t_1 (written t_2 \supseteq t_1)} \text{. Let } s = \{ \text{all } r \text{ and } t_2 \text{ } t \geq t \}, \text{ the relation } s \text{ can be found by applying a variation of the maybe select over the scheme for } t_2. \text{ For example, the query for } t_2 \text{ would be:}

Select from } r \text{ where:}

\[(A = a_1 \text{ or } A = \epsilon) \text{ and } (B = \epsilon) \text{ and } (C = c_1 \text{ or } C = \epsilon) \text{ and } (D = \epsilon) \text{ and } (E = \epsilon)\] .

The resulting relation, } r' = (r - s) \cup t_2 \text{ subsumes the relation } r_1. \text{ The relation } r \text{ has a larger possibility set, but the completions of } r' \text{ will all appear in the possibility set of } r. \text{ The second assumption allows the tuples in the current relation to be used to determine the true value of null values in the insert tuple, based on the existing set of functional dependencies. Null value replacement as discussed can be extended beyond just changing the insertion tuple. The defined values of the insertion tuple can be used to replace null values in the tuples of the current relation. Such a policy is clearly more time consuming, since, first it requires that the user interface (e.g. ASL compiler) generates test conditions for all functional dependencies for which the left side attributes of the inserted tuple are definite and then the definite right side attributes in the insertion tuple are used to replace nulls in the tuples of the current relation. The policy rests on the assumption that the insertion tuple is valid. Due to the associative nature of the operations in the database processor, one can modify the insertion algorithm to enforce the aforementioned assumptions effectively.

### 4.5 Evaluation

A distribution-driven simulator has been constructed to test the validity and feasibility of the proposed join algorithms. The aim of the simulator was to measure the relative merit of the three join algorithms with respect to execution time and space. As a basis for our simulation study, the following factors were used throughout the simulation:

i) Communication path between the preprocessors and join module is time shared and interleaved among the preprocessors.

ii) The number of preprocessors varies from 2 to 16 for the various simulation runs.

iii) Compaction factor is measured as the product of the horizontal and vertical (e.g. selection and projection) compaction factors of the tuples in a relation.

iv) Number of pages in the source and target relations varies from 10 to 100 for the various simulation runs.

v) Datablocks are of size 16k.

vi) Number of join attributes varies from 1 to 3.

vii) The timing and size performances are developed as average of quantities to smooth any anomalies which may occur due to databases with extremely poor or extremely efficient join criteria.

Simulation is based on time analysis of the basic operations as have been shown in Table 1.

ix) Simulator is constructed based on the parallel and pipeline nature of the proposed join algorithms.

Analytically and intuitively, several trends were expected in the results. For example, under the same conditions, true join would be expected to have a shorter execution time and a smaller resultant relation than the maybe join. The results for attribute maybe join should lie between those for true join and maybe join. Generally, as will be discussed later, the results bore out these and other expectations, and produced fairly constant relationships between the varying parameters and the execution time and size.

### Average Execution Time and Resultant Size

Entries in Table 2 depict the average execution time and size of the resultant relation for different join operations. As one could expect, true join screens out any tuples with null values on the join attribute(s), thereby eliminating a large number of join combinations and leading to a smaller execution time and resultant relation. Attribute maybe join, which allows some consideration of null valued join attributes, produced execution time and resultant relation size which were somewhat higher than those for true join, but also considerably less than maybe join.

### Effect of the number of join attributes

Table 3 depicts the average execution time and size of the resultant relation for different join attributes. It should be noted that this study did not take into account the inverse relationship between the join selectivity and number of join attributes. In
addition, attribute maybe join was tested with a constant number of join attributes and a varying number of required non-null join attributes. As can be seen, true join was almost unaffected by the number of join attributes. Maybe join shows a significant increase in both the execution time and the size of the resultant relation. This can be attributed to the increased number of searches and relaxed join conditions of maybe join. In the attribute maybe join, as the number of join attributes required to be total increased, the execution time and resultant size decreased slightly.

The simulation results have also shown that the execution time and the size of the resultant relation are directly related to the size of the source and target relations. In a separate simulation run, the sensitivity of the overall performance to the number of the

Table 2. Relative Performance of the Join Algorithms.

<table>
<thead>
<tr>
<th>Resultant Relation (Tuples)</th>
<th>C\text{Join} (\mu s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Join</td>
<td>24.3</td>
</tr>
<tr>
<td>Attribute Maybe Join</td>
<td>38.05</td>
</tr>
<tr>
<td>Maybe Join</td>
<td>57.75</td>
</tr>
</tbody>
</table>

relation size = 50 blocks  
# join attributes = 3

Each entry is the average of 200 simulation runs.

Table 3. Effect of the Number of Join Attributes

<table>
<thead>
<tr>
<th># of Join Attributes</th>
<th>Tuples</th>
<th>C\text{Join} (\mu s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Join</td>
<td>1</td>
<td>26.5</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>27.8</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>24.7</td>
</tr>
<tr>
<td>Attribute Maybe Join</td>
<td>3(1)</td>
<td>57.6</td>
</tr>
<tr>
<td></td>
<td>3(2)</td>
<td>57.3</td>
</tr>
<tr>
<td>Maybe Join</td>
<td>1</td>
<td>34.8</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>30.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>58.1</td>
</tr>
</tbody>
</table>

Relation size = 50 blocks  
Each entry is the average of 50 simulation runs.

communication paths between the secondary storage and join module was tested. As one can expect, we observed that the total execution time is very sensitive to parallel I/O operations.

5. Simulation to Determine Hardware Reconfigurability of ASLM

The modular design, replicated structure, and hardware independence of the components of ASLM provide a high degree of fault tolerance. In particular, components such as the secondary storage interface, the array of preprocessors and the database processor have been designed with the intent of creating a fault tolerant machine. Features such as the horizontal and vertical concatenation of the associative modules, allow the design to incorporate the fault tolerance of the components at the point of query initiation. A more appropriate strategy would allow reallocation of the modules within a component on a dynamic basis. Currently, we are looking at schemes to allow dynamic rescheduling of the modules in the secondary storage interface, the array of preprocessors and the database processor. In addition to the issue of fault tolerance, in a highly modular design such as ASLM, the issue of optimal use of machine resources is an important question. The single user environment of ASLM described in the previous sections doesn't lend itself to a high level of module utilization.

To improve the utilization, it is necessary to expand ASLM to a multiuser/multiprogram environment. It is clear that the dynamic rescheduling issues mentioned in the previous paragraph are closely tied to such an extension of the user environment. We have developed algorithms which allow the dynamic distribution of the resources in the secondary storage interface, the preprocessors, and the database processor among a set of atomic queries (we use the term atomic query to denote the request to load a relation). To test the feasibility of our algorithms based on performance parameters of the architecture, we have conducted an initial series of simulations. As a basis for our simulation study, we defined a set of 10 relations with varying parameters (e.g. size, number of attributes, etc.). An atomic query generator was used to determine which relation would be loaded and the complexity of the preprocessor operations during the load. Three levels of complexity were introduced into the process.

To consider the impact of different levels of preprocessor performance, we chose to run the simulations using existing microprocessors to represent the preprocessors. The simulations assume 128 cells in the secondary storage interface and the number of preprocessors were varied from 1 to 32 for the various simulation runs. Using instruction timing charts for the three processor types (MC 68020, Intel 80286 and Z 8000), an average processing time was calculated analytically for processing a block of data (Table 4).

Table 4. Average Execution Time for a Block of Data.

<table>
<thead>
<tr>
<th>Preprocessor Type</th>
<th>Average Execution Time (ms/block)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16 Kbytes/block</td>
</tr>
<tr>
<td>MC 68020</td>
<td>13</td>
</tr>
<tr>
<td>Intel 80286</td>
<td>33</td>
</tr>
<tr>
<td>Z 8000</td>
<td>53</td>
</tr>
</tbody>
</table>
Some preliminary results for the three preprocessors are shown in Figures 2, 3 and 4. Three types of information can be obtained from these data: First, by examining the data in Figure 3, one can see the impact on performance of our allocation strategy algorithms as the number of resources is increased. Secondly, comparing the entries in the Figures illustrates the impact of preprocessor speed on system performance. As one would expect, throughput is sensitive to both the number and the relative speed of the preprocessors. In our simulation, the processor speed played an important role in determining the number of simultaneously running queries. Increased preprocessor performance not only meant that a block of data could be processed faster, but it also meant that an atomic query seized fewer preprocessors during the load resulting in more atomic queries running at a time. Thirdly, because of the communication bottleneck between the secondary storage interface and preprocessors, after a certain point the preprocessors’ utilization will drop.

While our results are preliminary, they have been very encouraging. Our allocating algorithms have been proven to be correct and the work allowed us to develop a strategy for dynamic allocation of basic components in secondary storage interface, preprocessors and database processor to increase the fault tolerance capability of ASLM.

6. Conclusions and Further Remarks

The architectural feature of ASLM, a backend relational database machine was discussed. ASLM offers solutions to the so called semantic and computation gaps, and name mapping resolution and data communication problems. In addition, its architecture is designed according to the constraints imposed by the current advances in storage and device technologies. ASLM is functionally complete - it is capable of efficient handling of relational and update operations on unordered data files in associative fashion.

We have shown that ASLM is capable of handling an extended version of the relational operators based on maybe algebra operators without any overhead on the performance of the true algebra operators. Finally, ASLM has built-in capability for handling database management functions such as security validation, integrity control and recovery processing.

To improve the: i) fault-tolerance capability, ii) throughput and, iii) resource utilization of ASLM, we have developed a set of algorithms which allows dynamic reconfigurability of the resources in a multiuser environment. Our simulation results have shown the validity of these algorithms.

While these results are encouraging, one should study the effect of a more sophisticated communication networks between the secondary storage interface and the array of preprocessors. Moreover, due to the limited functionality of the preprocessors one has to determine the overall effect of special purpose architecture as a preprocessor. Finally, one should study the design and implementation of a specialized operating system and its effect on the system’s throughput.

Acknowledgement: This work has been supported in part by the Ben Franklin Challenge Grant and Locus, Inc.

References


Figure 2. Average execution time of the join operation for different types and number of preprocessors.

- Each entry is an average of 150 simulation runs
- Number of auxiliary storage = 2 units
- Number of communication lines between preprocessors and secondary storage interface = 2
- Number of communication lines between secondary storage and secondary storage interface = 2
- Join selectivity = 0.1
- Preprocessing selectivity = 0.01

Figure 3. Total execution time of the join operation for different type and number of preprocessors.

Figure 4. Concurrency vs. Hardware Resources.

- Type of Preprocessor: MC68020
- O: Average number of concurrent subqueries in preprocessors.
- X: Average number of concurrent queries in database processor.