MOBY: An Architecture for Distributed Expert Database Systems

Jonathan Bein
Roger King

Computer Science Department
University of Colorado
Boulder, Colorado 80309

Nabil Kamel

Computer Science Department
Michigan State University
East Lansing, Michigan 48824

ABSTRACT

In this paper, we consider MOBY, a distributed architecture to support the development of expert database systems in a rule based language. It combines standard indexing and horizontal data partitioning techniques with a rule based interpreter to achieve the reasonable performance. The major difficulty in developing this architecture is to maintain a high effective parallelism as the number of processors increases. Analytic results suggest that when data is reasonably well balanced across a local area network, MOBY has a high effective parallelism. Simulation results support this claim by showing that the effective parallelism is proportional to 40% of the number of processors. A discussion of some crucial issues in our current network based implementation is also given.

1. Introduction

Recent interest in expert database systems has stimulated research that combines techniques from artificial intelligence and database management systems(see [KERSCH85]). One branch of this research attempts to address the issue of handling large amounts of data in a rule based system ([BROD85, MOTO81, ZARRI84]). The problem is that traditional database query languages are restricted in the range of expression necessary for intelligent reasoning[HELD87]. Rule based systems, on the other hand, have historically been limited to handling small amounts of data because of their core memory orientation.

While there are many high level language issues concerning rule based systems, our approach in this paper is to assume that they are a close approximation to the type of language needed for writing an expert database system [1].

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

This work was supported by ONR under contract number N00014-86-K-0054.

[1] Strictly considered, Prolog is a logic programming language however it may also be viewed as a rule based language or production system.

This assumption is based on the combined power of the pattern matching primitives for database operations and the control constructs necessary for problem solving. Another assumption is that the requisite speed in a rule based system will not come from either the standard linear speedup obtained every few years in uniprocessors nor will a specialized uniprocessor suffice[QUIN85]. A final assumption is that associative memories are not yet cost effective to allow large systems.

Given these assumptions, our objective is twofold. First, we seek to apply standard database techniques to the management of large volumes of data in an expert database system. Second, we seek to map a rule based system and its data onto a local area network(LAN) architecture. The mapping we choose places a copy of the rule base on each processing element (PE) of the network. Execution proceeds concurrently on each PE. Intermediate results from execution are kept on PEs according to a horizontal partitioning scheme(see [SACC85, CERI83]). This mapping is a form of data parallelism[OZKAR86] where data is partitioned across processors. In functional parallelism [SHAW85], procedures or rules are partitioned across processors. Functional parallelism addresses high rules to data ratios. In contrast, data parallelism addresses high data to rules ratios.

MOBY is derived from OPS5[FORGY79] which has been the implementation language for several notable expert systems(e.g. [MCDM80 and KOWA83]) as well as the basis for study in parallelizing production systems[GUPT83, SHAW85]. Its efficiency, relative to other production systems, is derived from two sources. First, only a small fraction of the database is updated when a rule fires, thus the system can reduce much of the overhead in database lookup by remembering the state from one rule firing to the next. Second, queries in the different rules are frequently similar; hence, techniques analogous to multiple query processing[JARKE84] may be applied at rule compilation time to reduce the cost of redundant queries. The net effect of this technique is to save the results of previous joins. When a new tuple is inserted in a relation, as the result of a rule firing, any joins previously performed with this relation are incrementally updated. Until recently, these techniques have only been applied to databases operating in memory using linear search or hashing.

Notice that in the process of performing one incremental join, each newly joined tuple may be incrementally joined with the results from other previous incremental joins. Thus, several joins may be performed simultaneously at different parts of the LAN. Much work has been devoted to optimizing the execution of joins in a distributed environment(see [OZKAR86].) Our primary task is to exploit the potential for concurrency in a *rule based context*. Analytic results suggest large performance gains occur when indexing techniques and horizontal partitioning are combined with an incremental join

strategy. The main result of this paper is derived from a simulation of MOBY which shows that an incremental approach may be used effectively in join intensive rule based applications. When data is reasonably well balanced across the network, the rate of effective parallelism is proportional to 40% of the number of processors.

The next section includes background on the compilation of rules into a Rete network. This dataflow network reduces the overhead of database lookup and minimizes the cost of redundant queries. Section III provides an in depth coverage of an algorithm which drives the data parallel architecture of MOBY. Section IV covers a formal analysis and simulation results. Section V covers related work in database and expert system research. The last section looks at some additional research issues and conclusions of this study. An appendix with background on production systems is also provided.

2. The Rete Network

The objective in this section is to provide background on the operation of the Rete network as presented in [FORGY79]. The terminology has been recast from production system terms into database terms where possible. Readers unfamiliar with the use of production systems and OPS5 may turn to the appendix, although a broader discussion of production systems is in [BROWN85]. The focus here is on the compilation of production rules. Before considering the compilation strategy, it is important to note that these strategies were designed for in memory databases. From an artificial intelligence standpoint, "database" refers to a collection of facts, not a method of storage management.

The Rete network is the result of compiling a set of productions similar to the parse trees generated in database queries. Informally, we will see that, any condition element implies a relational selection. A join is expressed by using the same variable in more than one condition element.

In the recognize-act cycle, the recognize part dominates the processing time. In a naive scheme for pattern matching, on each cycle, all instances are matched against each condition element in each rule. The naive scheme is clearly prohibitive for large databases. The Rete algorithm was designed for efficient matching of a production by taking advantage of two characteristics of the database:

(1) *temporal redundancy* - A large percentage (more than 90%) of the database remains unchanged from one cycle to the next, hence query efforts can be saved. Most conventional databases have this characteristic too; e.g. an employee's salary will not change frequently [2].

(2) *pattern similarity* - Condition elements from different rules have a large amount of overlap, hence the matching of these can be performed simultaneously.

Figure 2 contains an annotated abstraction of the Rete network resulting from the compilation of our sample productions. It has two parts:

(1) A *selection network* which selects instances from the database according to a condition element and stores the result.

(2) A *join network* which joins the outputs from the selection network such that variable bindings are consistent. The results of the join are stored.

Even though the rules contain a total of four condition elements, they result in two main branches in the selection network: one for each data type. Because the first condition element from each rule is

[2] Conversely, for data that is changing in real time, as considered in [BEIN84], this technique will not suffice.

```
(p suggest_accountant_raise
(goal OBJECT raise-salary PERSON <n> STATUS active)
 ; Select an active goal instance for raising a salary
(employee NAME <n> SALARY {<salary> | <salary> < 27000}
 ; Select an employee instance whose earns less than $27000
  DEPARTMENT accounting) ; who is an accountant
-->
(write (crlf) Accountant <n> needs a raise.))

(p suggest_engineer_raise
(goal OBJECT raise-salary PERSON <n> STATUS active)
 ; Select an active goal instance for raising a salary
(employee NAME <n> SALARY {<salary> | <salary> = 35000}
 ; Select an employee instance who earns $35000
 DEPARTMENT engineering) ; who is an engineer
-->
(write (crlf) Engineer <n> needs a raise.))
```

Figure 1 - Example of Production Rules

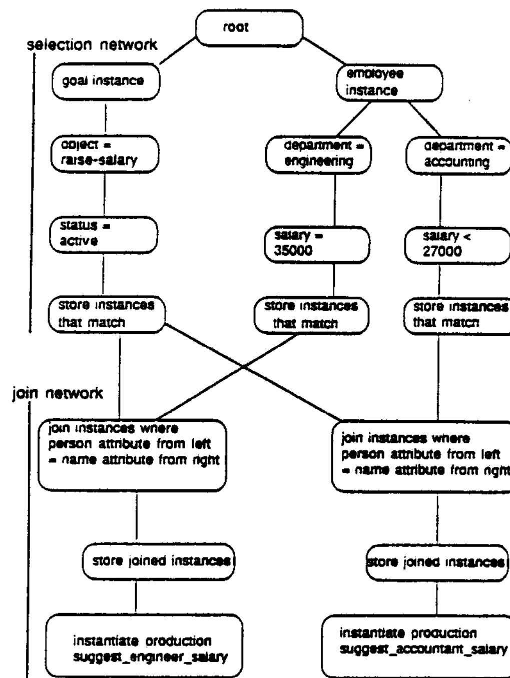

Figure 2 - The Rete Network

similar (in fact identical) the matching in the selection network may be shared. At the bottom of the selection network are *a-mem nodes* for storing tuples that have met the selection criteria. The a-mem nodes are the first place in the join network where the system takes advantage of temporal redundancy. Once an instance matches, it remains in an a-mem node until it is removed or altered.

In the join network, tuples whose variable bindings combine successfully with other tuples are stored. The nodes in this part of the network have two inputs: a left input and a right input. When a tuple arrives at either side, an attempt is made to join it with the tuples on the opposite side. Tuples that have been joined in this fashion are stored in *b-mem nodes* and may be combined with other a-mem nodes. This method of storing data in the b-mem nodes is the other way to take advantage of temporal redundancy: previously performed joins do not need to be recomputed. In Figure 2 there are two b-mem nodes in the join network. When the output from these

nodes arrives at the bottom of the network, a production is instantiated.

In summary, a production system has a fact database. The a-mem nodes stores selections performed on the database. The b-mem nodes stores joins. From herein, the term database will refer to data stored on disk.

3. Architecture and Algorithm

In this section we describe an algorithm to execute selections and joins in MOBY. The hardware configuration is described, followed by various parts of the algorithm expressed in pseudo-code.

3.1. Configuration

MOBY uses a LAN which consists of a control unit (CU) connected to a set of processing elements(PEs). The CU is responsible for the control of the recognize-act cycle, including synchronizing and handling communication between the PEs. Each PE has a local primary and secondary memory. The primary memory of each PE is initialized with a copy of the Rete network which encodes the rulebase. Portions of the database are placed in the memory nodes(a-mem and b-mem) of each PE. Also, each PE has a buffer to receive command messages from the CU.

This high level description has one very important requirement: the CU must be sufficiently powerful and the network sufficiently fast to guarantee that a PE rarely waits for a message. While precise requirements for other hardware components are still being determined, we do not anticipate the need for specialized hardware. Confidence in this estimate stems from a simulation on a powerful, but conventional workstation. A network of similarly configured systems should suffice. Also, the I/O requirements are moderate. For example, the formal analysis(below) is geared towards a disk with a total seek time of 10 msec.

To facilitate discussion, we make a distinction between a logical node in the network and an actual node in the network: conceptually, a logical node contains a whole relation, whereas an actual node contains a horizontal fragment of that relation. Thus, for each logical node, the number of actual nodes is the same as the number of PEs. The algorithm to perform distributed queries uses a horizontal data partitioning scheme. It attempts to store data with similar key values on the same actual node. The objective in partitioning is to balance the processing load. It is possible, though, that an actual node has no data either because their is not much data in the logical node or because the horizontal partitioning did not work well.

3.2. Control Unit Operation

During the act portion of the recognize-act cycle, the CU has the role of ensuring proper synchronization between actions and determining how to partition relations across the PEs. The partitioning occurs when messages are sent to the CU containing a recently joined (or selected), but not yet stored instance. The CU uses horizontal partitioning techniques to determine which PEs should get a copy of the instance and then sends a message to those PEs. Messages in this algorithm consist of a sender, receiver, node-number, and instance. The node-number is the logical node where this instance is to be stored. Because the CU does not require an acknowledgement from the receiver of the message, execution may continue after it sends the message. In the process of executing one message from the buffer, other messages may be written to the end of the CU buffer. The next action may not occur until all PEs are finished sending messages. This is determined by a timeout from each PE indicating that it has no more messages to send or process. This constraint is imposed to guarantee the correct serialization of actions in OPS5. [3] The algorithm for the act part of the recognize-act cycle is illustrated in Figure 3.

3.3. Processing Element Operation

The main task for each PE is to incrementally join an instance sent in a message from the CU. The execution consists of mapping the logical node contained in the message to an actual node on the machine. Once the mapping has occurred, a call is made to join the instance with existing instances on the node.

A PE must wait until each incremental join has completed before starting on the next message. In theory, this constraint could be relaxed and we could allow a multiprogramming or multiprocessing approach. Under the multiprogramming approach, the execution of incremental joins would be scheduled by the PE to maximize the consistent and frequent output of newly joined tuples. In turn, these tuples are fed to other PEs executing an incremental join. The exact potential of this tactic as it applies here is undetermined.

3.4. Storage Node Operation

Each storage node is implemented with a separate b-tree to contain the instances. An incremental join is invoked by executing a message that first stores a new instance in the b-tree for the node. Then the new instance is joined with instances from the opposite node.

```
procedure execute-actions(CU, actions)
control-unit CU;
list of action actions;
while actions do
 begin
 execute(action, random(CU.PEs));
 */execute the action on an arbitrarily chosen PE*/
 while CU.buffer do
 */continue message passing until the buffer is empty/*
  begin
  message := pop(CU.buffer);
  */remove the first message in the buffer/*
  receivers := horizontal-partition(message);
  */determine which machines will store the data/*
  for receiver in receivers do send(message, receiver);
  */send a message to be executed by each receiver/*
  end
 end
```

Figure 3 - Control Unit Algorithm

```
procedure execute-messages(PE)
processor-element PE;
begin
while PE.messages do
 begin
 message := pop(PE.buffer);
 */remove the first message in the buffer/*
 instance := message.instance(message);
 */extract the instance from that message/*
 node := logical-to-actual-node(message);
 */get the actual node corresponding to the logical node/*
 incremental-join(instance, node);
 */execute the incremental join/*
 end
end
```

Figure 4 - Processor Element Algorithm

[3] Actually, if a set of actions contains only insertions, then they may all be executed at once.

Each old instance that successfully joins with the new instance is sent immediately to the CU to determine its appropriate location. Clearly, it is possible to include more than one new instance with each message to be sent to the CU. We chose this minimal packet size to simplify the study.

Instances in this algorithm are indexed according to the variables that are being joined. In our example from Figure 2, the data from one side is indexed on the PERSON attribute; data from the other side is indexed according to the NAME attribute. Notice that this choice of key values for indexing in a node is defined by the rule, not by key fields of the relation type. The key fields for a given type as defined by the administrator may or may not overlap with their use in rules. The data is stored in each actual node as a separate b-tree. This storage is optimized to permit efficient retrieval of multiple instances per key value since that is the normal case.

3.5. Subtleties in the Algorithm

The quality of the horizontal partitioning is extremely important in this algorithm. In situations where the distribution is uneven, performance will suffer. Also, the horizontal partitioning strategy should maximize the likelihood that the output of an incremental join will not be be stored on the PE where it was derived. When this can be achieved, a greater degree of concurrency across the PEs will result because a receiving PE can process this new instance while the corresponding sending PE finishes its current work. This pipelining effect is a crucial property of the algorithm. The ultimate success of the horizontal partitioning function is data dependent, as such, the objective of reasonable balance cannot always be met. As mentioned above, this pipeline requires a sufficiently fast CU and network to guarantee that PEs do not wait long.

Programmer oversight or carelessness is another situation the partitioning must handle. For example, in the production below(Figure 6), although each condition element has one variable, it is not a common variable. This amounts to a join with zero variables, i.e. the cross product. This circumstance may be detected at compile time. The system responds to this situation by copying each instance to several actual nodes instead of one. The actual nodes are chosen to guarantee that each instance from both relations will combine exactly once. The efficiency of this operation is discussed in the next section.

```
procedure incremental-join (new-instance, node)
storage-node node;
begin
store-instance(new-instance, node);
instance new-instance;
memory-node node;
*/insert the new instance into the b-tree for this node/*
key := key(new-instance);
old-instances :=
 lookup(key, opposite-node(node));
 */from the opposite storage node,
  get all the instances that join with the key/*
for old-instance in old-instances send(join(instance, old-instance), CU);
 */combine each old-instance with the new-instance
 as a message to the control unit/*
end
```

Figure 5 - Incremental Join

```
(p cross-product
  (type-1 ^ATT-1 <var-1>)
  ; select instances of type-1 and form a binding for the first attribute
  (type-2 ^ATT-2 <var-2>)
  ; select instances of type-2 and form a binding for the second attribute
 -->
```

Figure 6 - A Cross Product

4. Analysis and Simulation Results

In this section we consider a formal analysis of MOBY. The formal analysis covers a best, average, worst, and probable worst case scenario. These scenarios are classified by their performance on the incremental join, but analyzed for the rate of effective parallelism. Following the analysis, simulation results for an entire system are presented. These results are mapped onto the more granular predictions from the formal analysis and reviewed for discrepancies. Interpetation of this data suggests that the architecture we have developed needs a sufficiently high number of joins to exploit potential concurrency.

4.1. Formal Analysis

The analysis of a data flow network is difficult because it is highly data dependent. Also, the analysis of distributed systems is difficult when the level of synchronization is low. In this section we make some simplifying assumptions to permit analysis. While much database research may overemphasize worst case analyses[CHRIST84], this analysis may be faulted for focusing too much on the average case. In defense, there are adaptive components to the algorithm which do tend towards the average case. First, we present the analysis and then give an example of the expected performance.

The analysis is oriented to reflect the quantity of tuples that successfully join in any actual node. Within that dimension, the objective is to optimize processing at an actual node. The cost of processing at that node is measured mostly by disk access time, although the amount of communication is quantified. The parameters for our analysis are described below:

> R - The cost of retrieving a tuple from the disk. Even though the tuple may already be in memory as a result of cacheing, for the analysis we assume it is on disk.
> M_i - The number of instances at a logical node i,
> m_{ij} - The number of instances at actual node i on processor j.
> k - The size of an index node in a b-tree.
> e - The size of an data block in a b-tree.
> H_i - The number of instances resulting from a join or the number of "hits" at logical node i.
> h_{ij} - The number of instances resulting from a join or the number of "hits" at the actual node corresponding to logical node i on processor j.
> D_i - The number of possible values of a domain for the attribute being joined at logical node i.
> n - The number of PEs in the LAN.

The time to lookup a set of instances that have a given key value is based on the standard b-tree lookup analysis [ULLMAN82]. Using the parameters from above, the time is proportional to $(\log_k (m_{ij}/e))*R$. Because additional instances are stored contiguously in our b-tree implementation, the time is given by $(\log_k (m_{ij}/e) + h_{ij}/e)*R$. The cost of the incremental join will be reflected indirectly by *Hsubi*. H_i does not measure the time spent at one actual node, rather, it measures how much work this node creates for the rest of the system. Given these parameters there are four cases to analyze.

4.1.1. Case 1 - Best Case

The best case time for this algorithm is when h = 0. Then the cost is only proportional to the lookup time and nothing else. However, this results in low processor utilization and therefore zero effective parallelism.

4.1.2. Case 2 - Probable Worst Case

Informally, one can see that when h is large, system performance may suffer. Specifically, when h = m each instance from one relation joins with every instance from the other. This occurs when, as discussed above, lack of a common variable in condition elements results in a cross product. However, MOBY recognizes this and partitions data so that each node contains an amount of data proportional to $\sqrt{n}$. This is the probable worst case because it arises in practice for intentional or unintentional reasons.

4.1.3. Case 3 - Worst Case

Other cases occur where a rule does not appear to result in a cross product. The condition elements have a common variable in this situation, but, coincidentally each instance from one relation joins with all instances in the other relation. Under this circumstance where h = m, one machine would end up performing the entire join for a given relation instead of distributing the work across the machines in the network. This situation cannot be detected at compile time. Degradation will result in performance proportional to execution in a uniprocessor environment. Intuitively, it seems unlikely that a large set of instances would all have the same key value. Hence, the previous case is considered the probable worst case.

4.1.4. Case 4 - Improbable Average Case

The value of h we use for the average case is hs for simplistic h. For an arbitrary node i, $hs_{ij} = m_{ij}/(n * D_i)$. This value for h assumes an even distribution of the values of a domain across the instances. For each additional field that is joined, hs_{ij} is divided by a number similar to D_i. This case is improbable because domain values may not be uniformly distributed and partitioning may not be even. This case is included to provide a sense about performance under the maximum effective parallelism.

As with many synchronized distributed algorithms, time is proportional to the slowest element. In our algorithm, the unit of synchronization is an action (as opposed to the recognize-act cycle). Therefore, the time to execute an update is also proportional to the speed of the slowest PE. When data is appropriately balanced the distribution for the domain of each attribute is uniform, each PE has a nearly uniform execution time so that the overall speed is proportional to the number of PEs. Table 1 summarizes the analytic results for a possible configuration. The configuration varies the number of PEs while using the following constants; R = 10msec, M_i = 100,000, k = 127, e = 10, HS_i = 2,500.

As the number of processors increases, the initial descent during lookup starts to dominate the cost of the join. In effect, the cost of looking up multiple instances is spread across the PEs. Consequently, the number of processors should be adjusted so that lookup time does not dominate retrieval and so that the total time to join is less than some requisite constant. These numbers are indicative of very high performance and would permit the use of much larger expert database systems than currently exist.

PEs	Lookup	Retrieval	Total
1	15	2500	2515
5	15	500	515
10	15	250	265
100	15	25	35

Table 1 - Predicted Times

Contrary to expectation, the approach used in MOBY does not necessarily tradeoff performance for storage space. An approach which uses pointers on disk may be combined with a lazy evaluation scheme. Under this approach, unique identifiers are assigned to each working memory element. The output of selections and joins stores only the unique identifiers of these working memory elements. When particular attribute values are needed, the actual working memory element is dereferenced. This scheme is currently under implementation.

4.2. Simulation

The logical simulation we implemented was conducted to determine the effective parallelism of the distributed incremental merge. It involved a modified OPS5 interpreter with 3000 lines of Lisp source code for the simulation. No attempt was made to determine communication costs. Work was performed on a Symbolics 3640, using a Winchester 167.5 megabyte disk drive, with 4 megabytes of main memory. Processes on the machine were minimized to prevent undue interaction with the paging system. For realism, garbage collection was turned on during our simulation although it is less efficient. Finally, only insertions were used instead of deletions or updates.

We developed an implementation of a b-tree algorithm on the Symbolics to demonstrate efficient retrieval, however, it was not incorporated into the simulation. As such, the simulation reflects sequential search in performing joins.

4.2.1. Simulation of Case 3 - Probable Worst Case

The first set of experiments used a rule(Figure 7) that resulted in the probable worst case. The salient feature of this example is that the condition elements have no common variable. Also, the action is guaranteed to create new instances that join with existing instances. In the experiment, most data was generated by a program. This rule results in a large amount of incremental joins. The times in msec obtained for one, four, and nine processors were 225, 139, and 81 respectively. The effective parallelism for four processors was .404. For nine processors, the effective parallelism was .303. We use this probable worst case to show that a reasonably high effective parallelism can be obtained when there is sufficient join potential. For a given set of data, there is definitely a point beyond which additional processors are not useful. The decrease in effective parallelism from four to nine processors reflects that trend.

4.2.2. Simulation of Case 4 - Improbable Average Case

The second set of experiments used software to generate all rules and data. The parameters to the system allowed us to configure different rule bases. They varied by the number of rules, the grain size of each rule, the amount of data, and the probable number of hits.

```
(p R1
  (type-1 ^ATT1 <var-1> ^ATT2 <var-2>)
  ; Select instances of type-1 and form bindings for the 1st and 2nd attribute.
  (type-2 ^ATT1 <var-3> ^ATT2 <var-4> ^ATT3 9)
  ; Select instances of type-2 and form bindings for the 1st and 2nd attribute.
  ; The third attribute must be 9.
  -->

  (make type-1 ^ATT1 (compute 1 + <var-1>) ^ATT2 <var-2>)
  ; insert a new type-1 instance
  (make type-2 ^ATT1 <var-2> ^ATT2 (compute 1 + <var-1>) ^ATT3 9))
  ; insert a new type-2 instance
```

Figure 7 - Rule for Probable Worst Case Scenario

The distribution of domain values for various attributes and the variation of data were nearly uniform. In contrast, the rules were not generated to be as homogeneous. The configurations that were used varied between 50-150 rules, 5-8 relation types, and 500-5000 instances. The number of instances was kept small because of the high cost of simulation. Most of the simulations ran for exactly 500 rule firing cycles.

Table 2 summarizes the results we obtained. The times and effective parallelism were shown for one, four, and nine processors. The major conclusion to draw is that the a requisite join potential is a necessary condition for obtaining reasonable effective parallelism. The last three examples support this claim: there were very few hits and there was virtually no speedup. Under the right circumstances, this algorithm can obtain an effective parallelism proportional to .4 of the number of processors. This speedup does appear to data dependent.

4.2.3. Optimism and Pessimism in the Method

In this subsection, a critique of the experimental method is given. The critique covers aspects of the method which will make the simulation results look more optimistic as well as pessimistic.

communication overhead - Certainly, in a system implemented on a LAN, communication is an issue. However, in the data that we have observed, only 40,000 messages were sent in the worst case over a one-hour period. Further, the size of the messages is the size of an instance as opposed to the size of a large file. In the future, packets larger than an instance may be sent over the network, correspondingly, the number of packets will decrease. This is perhaps the weakest link of the study.

data quantity - The largest quantity of data used in this study was 5000 instances. Yet, the stated objectives of this study are to work with a much larger database. Moving to a full implementation on the network will allow more realistic databases.

unrealistic data - For this study, it is not clear what realistic data is. As yet, nobody has built an expert database system in OPS5. The most notable examples of OPS5 usage have a large number of rules and a small amount of data, so they do not provide a suitable testbed. The data was generated to exemplify reasonable variation and distribution according to adjustable parameters.

simulation - Although the simulation provided useful data, the level of granularity does not permit observation of some important interactions that will shed more positive light on the study. For example, the effects of pipelining during the incremental merge are not faithfully simulated. So, in fact, the effective parallelism may in some cases be higher than what was reported.

Case	Rules	Data	Hits	1PE	4PE	9PE
1	75	1690	6194	3671/1	2756/.333	1327/.307
2	100	2655	4150	2512/1	1387/.45	1067/.26
3	75	2454	9717	2125/1	1268/.41	904/.26
4	150	3781	25584	3995/1	3042/.32	1790/.211
5	150	2773	22613	2376/1	1552/.38	644/.437
6	150	3546	32582	2743/1	1803/.38	1491/.204
7	100	3151	6976	306/1	209/.366	182/.18
8	75	3510	3554	107/1	114/0	86/.13
9	50	1975	2418	53/1	50/0	52/0
10	75	3352	5013	110/1	114/0	120/0

Table 2 - Simulation Results

join quantity - An assumption is that the join quantity needs to be high enough to justify the use of this architecture. In all of the experiments, the actual hit ratio was still fairly low, approximately 15 per rule firing. Even in the probable worst case study, the number of hits was on the order of dozens per rule firing, rather than thousands. In real databases, a higher hit ratio will yield higher effective parallelism.

5. Related Work

There are two main strands of research related to our work. The first area is the development and use of expert system tools which mostly address language issues. The second area is the study of parallel architectures for production systems which mostly address implementation and performance issues. These are described immediately below.

5.1. Expert System Development Tools

The academic research in knowledge based systems of the 1970's placed much emphasis on finding the "right" rule based interpreter. Those systems typically employed one control strategy, one search direction, one inference mechanism, and one representation formalism. Early 1980's efforts to create hybrid systems employed multiple control strategies and representation formalisms (see [BOBR83].) The consensus was that no single approach was sufficient, therefore an integrated approach is best. Examples of these tools include Art[INFER84], Kee[INTELL86], and Knowledge-Craft[CGI85].

These languages are much better suited to building expert database systems than OPS5 because of their diversity. Although these tools tend to have the right language primitives for building an expert database system, they are not engineered to permit very large databases. Admittedly, both Kee and Knowledge-Craft allow queries to be submitted to a database, yet the basic operation of these tools is still oriented towards management of a small number of objects. Our work investigates a much tighter coupling.

5.2. Parallel Architectures for Rule Based Systems

Attempts to parallelize rule based systems have assumed that the ratio of rules to data is high. As such, they have concentrated on permitting large numbers of rules to run efficiently. In this section we discuss production parallelism. Under this approach, processors in a massively parallel system are allocated around productions. The best known work in this area is from Columbia University ([SHAW85 and STOLFO84]), where there have been several different formulations and implementations of production parallelism on NON-VON and Dado. These implementations all have several features in common:

(1) The system configuration is a MIMD based tree structured machine consisting of many small PEs.

(2) The PEs use associative memories to support the join operation.

(3) The allocation of processors is based on rules rather than data. This may mean allocating one node in the Rete network to a PE, allocating one rule to a PE, or allocating several rules to a PE. This is the distinguishing characteristic of production parallelism as compared to data parallelism.

The different implementations were analyzed for performance on six well known expert systems[GUPT83]. The best performance achieved about 900 firings per second. This approach of distributing productions to processors, however, has a limited potential for improving performance even in systems that are knowledge intensive. This point was shown by Oflazer in [OFLA84], who attributes

it to the fact that in most production systems a production, when fired, may only affect a small number of other productions (about 35 on the average).

For expert database systems, there are two problems with architectures with the properties just described. First, at most 35 fairly simple processors may simultaneously process any one memory element. In MOBY, as few as one production may be affected, yet a large number of PEs are active in response. Second, the use of associative memories is not feasible mainly because of size. Real systems may have 100,000 instances stored at one memory node. Hence, traditional database methods are necessary in this situation.

6. Future Directions and Conclusion

There are several areas that we have not covered in our research that are related to achieving better performance.

The algorithm for MOBY relies on a reasonably well balanced partitioning. Although the algorithm attempts to prevent the storage of an instance on the machine where it was derived, there is no adaptive approach for re-arranging data within a logical node that is imbalanced. This type of adaptation may prevent degradation resulting from inappropriate partitioning.

Standard query optimization techniques for multiple joins attempt to perform them in an order that involves the least amount of data first. In the Rete network, the order of performing joins is based on the clause ordering by the user. Already, MOBY detects the cross product at compile time. The execution of the cross product may be delayed or prevented if the system reorders the clauses. Aside from this simple optimization, reconfiguration of the network to reorder joins is expensive. In short, the advantage of storing the results of joins has a tradeoff if the quantity of data does not match expectations.

Conflict resolution is one of the defining components of a rule based interpreter. The intent is to "resolve conflicts". Unfortunately, in an expert database system there may be many instantiations and thus many conflicts. Large conflict sets may be difficult to manage and incorrect behavior may result. In future research, we hope to find some way of streamlining the conflict resolution process when it becomes large.

Another area of future research combines functional and data parallelism into one architecture. One problem with functional parallelism is that all processor allocation is performed at compile time, i.e., it is static. Conversely, data parallelism mostly addresses runtime characteristics of the system, i.e. it is dynamic. Based on work in [GUPT84], a large percentage of condition elements in a system cluster around a small percentage (< 30%) of relation types. Such information can be obtained at compile time. There is other information relating to intelligent allocation of processors that is available at compile time.

We are currently implementing MOBY on a LAN consisting of nine Symbolics workstations. Moving from simulation to the actual system will provide a much better experimental environment. For example, the high level requirements of the horizontal partitioning scheme have been established, but a true evaluation will be easier with a complete implementation.

We have seen that expert database systems have different characteristics from database and expert systems. The computational demands necessary to support intelligent database systems suggest that a uniprocessor will not provide adequate performance. Further, the inherent parallelism can not be exploited on a large mainframe. The compilation methods for rule base systems may work very well for join intensive systems. Combining these techniques into MOBY has three main results:

(1) a communication overhead proportional to the number of instances that join.

(2) a data throughput which permits the use of OPS5 style production systems to operate on databases which are far larger than those previously built.

(3) a speedup proportional to 40% of the number of processors for reasonably balanced data.

Acknowledgements

We wish to thank Brigham Bell for discussing some ideas pertaining to efficiency. Thanks also to Tom Rebman for work on the b-tree implementation. Finally, thanks to Hal Eden for providing a supportive computing environment.

References

[BEIN84] Bein, J. et al. FIES: An Expert System for Handling Spacecraft Hardware Failures. Proceedings, Conference on Intelligent Systems and Machines, Rochester, Michigan, 1984.

[BOBR83] Bobrow, D.G., and Stefik, M.L., The Loops Manual, Xerox PARC, 1983.

[BROD85] Brodie, M.L., and Jarke, M. "On integrating logic programming and databases" Proceedings, Conference on Expert Database Systems, Columbia, S.C., 1985.

[CERI83] Ceri, S., Navathe, S., and Wiederhold, G., Distribution design of logical database schemas, IEEE Transactions of Software Engineering 9, 4 (July 1983), pp. 487-503.

[CGI86] Carnegie Group Incorporated, Knowledge-Craft Reference Manual, Carnegie Group Incorporated, Pittsburgh, PA., 1986

[FORGY79] Forgy, C.L. "On the Efficient Implementation of Production Systems", Department of Computer Science, Carnegie-Mellon University, Ph.d Thesis, February 1979.

[BROWN85] Brownston, L., et al, Programming Expert Systems in OPS5: An Introduction to Rule Based Programming, Addison Wesley, 1985.

[CHRIST84] Christodoulakis, S., Implications of certain assumptions in database performance, ACM Transactions on Database Systems 9, 2 (June 1984), 163-186.

[QUIN85] Quinlan, J., "A Comparative Analysis of Computer Architectures for Production System Machines", Department of Computer and Electrical Engineering, Carnegie-Mellon University, internal report CMU-CS-85-178, May 1985.

[GUPT83] Gupta, A., and Forgy, C., "Measurements on Production Systems", Department of Computer Science, Carnegie-Mellon University, internal report CMU-CS-83-167, December 1983.

[GUPT84] Gupta, A., "Implementing OPS5 Production Systems on DADO", Department of Computer Science, Carnegie-Mellon University, internal report CMU-CS-84-115, March 1984.

[HELD87] Held, J.P., and Carlis, J.V., MATCH - A new high-level relational operator for pattern matching, ACM Communications 30, 1 (January 1987), 62-74.

[INFER84] Inference Corporation, ART Reference Manual, Inference Corporation, Los Angeles, CA. 1985.

[INTELL86] Intellicorp, KEE 3.0 Reference Manual, Intellicorp, Mountain View, CA., 1986

[JARKE84] Jarke, M. and Koch, J., "Query Optimization in Database Systems", ACM Computing Surveys, June 1984.

[KERSCH85] Kerschberg, L., Second Conference on Expert Database Systems, Charleston, South Carolina.

[KOWA83] Kowalski, T. and Thomas, D., "The VLSI Design Automation Assistant: Prototype System", in: *Proceedings of the 20th Design Automation Conference*, ACM and IEEE, (June 1983).

[MCDM80] McDermott, J., "R1: A Rule Based Configurer of Computer Systems", Department of Computer Science, Carnegie-Mellon University, internal report CMU-CS-80-119, 1980.

[MCDM82] McDermott, J., "XSEL: A Computer Salesperson's Assistant", In J.E. Hayes, D. Michie, and Y.H. Pao, Machine Intelligence, Horwood, 1982.

[MOTO81] Moto-oka, T., Fifth Generation Computer Systems, North Holland Press, 1981.

[OFLA84] Oflazer, R., Partitioning in Parallel Processing of Production Systems, IEEE Proceedings, 1984.

[OZKAR86] Ozkaharan, E., Database Machines and Database Management, Prentice-Hall Press, 1986.

[SACC85] Sacca, D. and Wiederhold, G., Database partitioning in a cluster of processors ACM Transactions on Database Systems 10, 1 (March 1985), pp. 29-56.

[SHAW85] Shaw, D.E. NON-VON's Applicability to Three AI Task Areas. Proceedings of the IJCAI, 1985, pp. 61-72.

[STOLFO84] Stolfo, J., Miranker, D., and Shaw, D. "DADO, A Parallel Processor for Expert Systems", in: *Proceedings of the 1984 International Conference on Parallel Processing*, IEEE and ACM, pp. 74-82, 1984.

[ULLMAN82] Ullman, J.D., Principles of Database Systems, Computer Science Press, 1982.

[ZARRI84] Zarri, G.P., Constructing and utilizing large fact databases utilizing artificial intelligence techniques, Proceedings of the First Internation Workshop on Expert Database Systems, Columbia, S.C., 1984.

Appendix - OPS5, Productions Systems

The objective in this appendix is to introduce definitions and notation pertaining to production systems, in general, and OPS5, specifically. OPS5 is a production system, which is a programming language consisting of three parts:

1. A *working memory* which is a global database of facts called working memory elements.

2. A *production memory* containing production rules that encode expert knowledge.

3. An *interpreter* that applies the rules to working memory in solving a problem.

A *rule* in production memory has a name, an *if* portion (also called the condition), and a *then* portion (also called the action.) The condition portion has *condition elements* which *match* or *query* the

wm.1 = (employee NAME Joe Jones DEPT accounting SALARY 26500)
wm.2 = (employee NAME Fred Blee DEPT accounting SALARY 25500)
wm.5 = (department NAME accounting BUDGET 500100)
wm.6 = (department NAME operations BUDGET 250000)
wm.7 = (goal OBJECT raise-salary PERSON Joe Jones STATUS active)
wm.8 = (goal OBJECT raise-salary PERSON Fred Blee STATUS active)

Figure 8 - The Database

employee: NAME DEPARTMENT SALARY
department: NAME BUDGET
goal: OBJECT PERSON STATUS

Figure 9 - Data Definitions

database. A condition element may contain *variables* or *constants* as shown in both rules in Figure 1. Variables are surrounded by "<" and ">" to distinguish them from constants.

The rules from Figure 1 might be used to suggest raises for employees in the accounting and engineering departments. Rule suggest_accountant_raise determines employees in the accounting department who are eligible for a raise, and also earn less than $27000. It will report the names of such employees on the terminal.

We illustrate the matching of rule suggest_accountant_raise to the database from Figure 8. The first condition element is a selection of any goal data type whose object is raise-salary and whose status is active, thus it matches wm.7 and wm.8 The second condition element selects any accountant whose salary is less than $27000; thus it matches wm.1 and wm.2. To instantiate the rule, the data which matches each condition element must meet the criteria necessary for a join: the variable <n> must be the same. We call this a *consistent binding*. In our small example, wm.1 and wm.7 form consistent bindings, as well as wm.2 and wm.8. Other algebraic constraints may be placed between bindings, such as <, >, <=, etc.

The interpreter applies these rules in a three step phase, collectively called the *recognize-act* cycle. The first part of the cycle matches the rules to the database and finds consistent bindings. Each set of instances that match and bind consistently to a condition is called an *instantiation* of that rule. The *conflict set* is the set of instantiations output from the match cycle. Next, the *conflict resolution* phase chooses one rule from the conflict set. Finally the act phase, applies the action of the chosen rule. Each part of the action may modify, remove, or add new working memory elements.