# Performance Evaluation of a Storage Model for OR-Parallel Execution of Logic Programs

by

Andrzej Ciepielewski and Bogumil Hausman

SICS R86003
Research Report
ISSN 0283-3638

# Performance Evaluation of a Storage Model for OR-Parallel Execution of Logic Programs

**Andrzej Ciepielewski**
**Bogumil Hausman**

Swedish Institute of Computer Science (SICS)
Box 1263, S-163 13 SPÅNGA, Sweden
Electronic mail: andrzej@sics.uucp or bogdan@sics.uucp
Phone: + 46 8 750 79 70

**Abstract**

As the next step towards a computer architecture for parallel execution of logic programs we have implemented four refinements of the basic storage model for OR-Parallelism and gathered data about their performance on two types of shared memory architectures, with and without local memories. The results show how the different properties of the implementations influence performance, and indicate that the implementations using hashing techniques (hash windows) will perform best, especially on systems with a global storage and caches. We rise the question of the usefulness of the simulation technique as a tool in developing new computer architectures. Our answer is that simulations can not give the ultimate answers to the design questions, but if only the judiciosly chosen parts of the machine are simulated on a detailed level, then the obtained results can give a very good guidance in making design choices.

**Keywords:** logic programming, OR-parallel execution, performance evaluation, computer architecture, simulation, benchmark.

**List of Contents**

## 1. Introduction

The designers of sequential computers for traditional languages like Fortran, Pascal, C, or even Lisp or Prolog, can rely on many years of experience in programming in such languages and on many measurements characterizing their performance, [1,2,3]. The situation is different in the design of parallel systems. There is very little data to rely on [4,5], and for a language like PROLOG there is virtually no data.

We are considering setting up an OR-parallel Prolog system on a parallel computer. OR-parallelism enables alternative solutions to a query to be found in parallel. One of the important differences between a sequential implementation of Prolog (depth-first with backtracking) and an OR-parallel one, is that in the latter there are several sets of bindings existing at the same time. Thus, a major implementation problem is to find a storage organization, which would replace the very efficient one used in today's systems [6] .

In this paper we present performance results for several storage structures on two shared memory architectures. The storage implementations range from a simple one, just providing separate logical address spaces for different branches in a search tree, to one using hashing techniques for accessing variables. Our results show, among other things, that the careful choice of implementation can yield many-fold difference in speed. They also show how the presence of a storage hierarchy changes performance of the different implementations.

The current work has been preceded by an investigation of the dynamic behavior of Prolog programs [7]. That investigation has helped us to choose a set of benchmark programs and provided us with some clues for understanding the obtained results. The work presented here can also be seen as a continuation of the work of Crammond [8], though it has been conducted independently. Crammond investigated three unrelated storage models in an implementation independent manner. We start from the storage model presented in [9] and proceed with several implementation oriented refinements.

The rest of this paper is organized as follows. In section 2 the basic process and storage models are shortly described. In section 3 the considered architectures are introduced. In section 4 the storage implementations are described. In section 5 the benchmark programs are shown. In section 6 the results are presented and discussed. Finally, in section 7 we give some conclusions, and outline our future work.

## 2. Process Model of Execution and the Basic Storage Model

Or-parallel execution of a program can be seen as a parallel traversal of the program's search tree by a set of processes. A node in a tree represents the state of a process. The state consists of a goal-list and a binding environment. Execution is initiated in the root of a tree by creating a process to execute the top-level query. A top-level query consists of one or more goals in a conjunction. Execution in the root proceeds by choosing the first goal in the process goal-list and creating a child

process for each clause with the name of the head matching that of the current goal. The parent process terminates. Each child process executes unification. In the course of unification the new variable bindings are added to the environment of the process. If the unification fails the branch is terminated. If it succeeds, the goals in the body (if any) are preappended to the current goal list. If the goal list is empty the branch is ready and can provide a result, otherwise execution proceeds as described for the root.

The binding environment of a process consists of contexts (activation records) for storing values of variables. A new context is created each time a clause is invoked. Our measurements have been done on an implementation using structure-sharing. In a structure-shared implementation the value of a bound variable consists of a pair of pointers, a pointer to the static representation of the term to which the variable is bound, and a pointer to the context in which the term is interpreted. An unbound variable is represented by the value UNBOUND.

The binding environment of a process is the process logical address space, which is separated from the address spaces of other processes. In [9] we have proposed a storage structure which ensures that each process has a separate logical address space, but allows sharing of some contexts. The storage in the proposed model consists of directories and contexts. A directory contains references to contexts. The binding environment of a process is represented by a directory and the contexts referred from it. Variables in the environment of a process are accessed via a unique name, a triple: <directory address, context offset, variable offset>. The directory address is the address to the base of a directory, the context offset is the offset from the directory base to the entry where the context address is placed, and the variable offset is the offset from the context base to the entry where the variable value is placed.

When a context is created on invocation of a clause, all its entries are initialized to NULL and its address is placed in the process directory. When a process creates two or more children processes, each process gets a new directory. Each new directory is created in the following way. Each context in the old directory is investigated. If a context does not contain unbound values, we say it is committed, the reference to it is placed in the new directory at the same offset as in the old one. If the context contains unbound variables, we say it is uncommitted, a copy of the context is made, and a reference to the copy is placed in the new directory at the same offset as in the old one. In this way each process gets private copies of uncommitted contexts, which ensures separation of address spaces, and all processes share committed contexts, which is possible because of the single assignment property of logic programming languages.

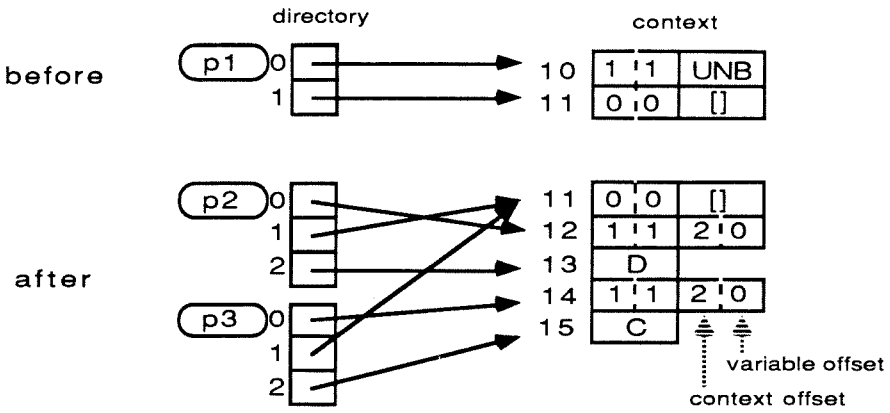



Figure 1.

Two snapshots of the storage content. In the first (before) there is one process, p1. The binding environment of p1 is represented by its directory, containing two entries, and the contexts 10 and 11 referred from entries 0 and 1, respectively. The context 10 is uncommitted, since it contains an unbound value UNB. The context 11 is committed. In the second snapshot (after) there are two processes, p2 and p3, which are children of p1. The binding environment of p2 is represented by its directory, and the contexts 11, 12, and 13. The binding environment of p3 is represented by its directory and the contexts 11, 14, and 15. The context 11 is shared between the environments. The contexts 12 and 14 have been created from the uncommitted context 10, and finally the contexts 13 and 15 have been created for the clauses invoked by p1 and p2. The variable which has been unbound in the environment of p1 (in the context 10), is bound to constants C and D in the environments of p2 and p3 in the contexts 12 and 14.

## 3. Multiprocessor Organizations

Implementation of logic programming systems on multiprocessor computers requires architectures supporting a global address space. A global address space can be provided on many multiprocessor organizations. We have estimated performance of our storage implementations on two contrasting structures characterized by Halstead in [10]: the "dance hall" model where a network connects processors to memory modules, and the "myriaprocessor" model, which features a myriad of nodes, each having processing and memory components, connected to each other in some topology. We assume that in either case the network is a simple shared bus. The memory modules of the dance hall configuration form a pool of globally shared memory, we think of a myriaprocessor as having a global shared memory distributed among the memories of various nodes. We assume, for simplicity, that the systems have no storage hierarchy. The "dance hall" and the "myriaprocessor" models are the extrems of a spectrum of existing architectures. Approximative examples of dance hall machines include NYU Ultracomputer [11], Multimax [12], and Balance8000 [13], examples of myriaprocessors include Cm* [5], RP3[14], and the BBN Butterfly machine [15].

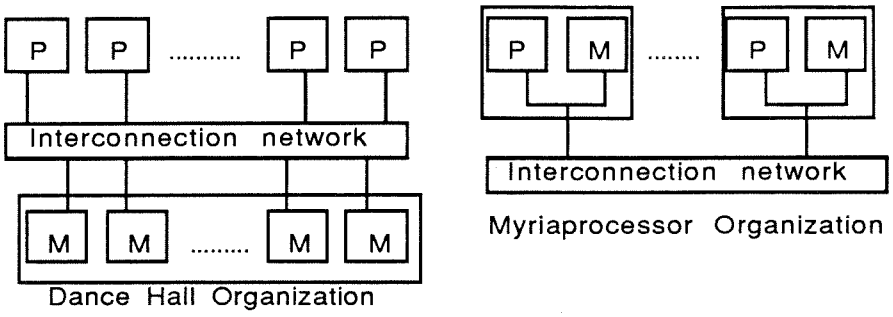



Figure 2.
Multiprocessor organizations.

To simplify of our investigation we make the following assumptions about the systems. First, a system consists of many processors (between 16 and 64) and the load is distributed equally among the processors, this assumption is important for definitions of locality in the next section. Second, a processor has a support for computing a hash address (a hash function) as fast as it produces an ordinary memory address, but has no special support for accessing the hash addressed objects in the memory. Third, accesses to the static data, and accesses to and processing of the instructions take the same time in both types of systems. Finally, the load distribution takes no time.

We define performance in a dance hall system as the number of accesses to the dynamic data, and in a myriaprocessor system as the sum of the numbers of local and non-local access to the dynamic data weighted by the factors specified below. We assume that initializations, and read and write operations on a word of storage take the same amount of time.

We use the storage access time in a dance hall system as the unit of time, and assume that in a myriaprocessor it takes 0.3 unit to access on-node storage (local access), and 1.3 unit to access remote storage (non-local access). The figures describe the ratio in a MC68020 system using the VMEbus [16, 17], for the Concert system described by Halstead et al. [18] the numbers are 0.5 unit, and 1.5 unit, respectively. The figures are realistic assuming there is no bus or memory contention.

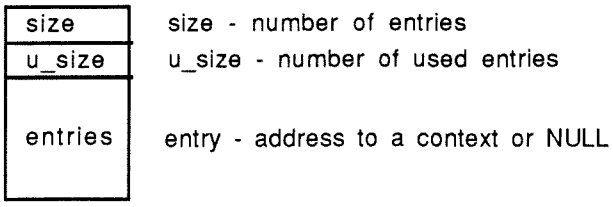
## 4. Storage Implementations

We have implemented four refinements of the basic storage model described in Section 2. The first one is a straightforward implementation of the basic model. In the second we introduce trees of directories and fetching of contexts on demand. In the third we limit the size of directories and introduce hashed access to contexts. In the fourth we do not use contexts explicitly, and instead apply hash addressing directly to variables. We try different copying strategies on all the implementations, and different sizes of hash windows on the last two.

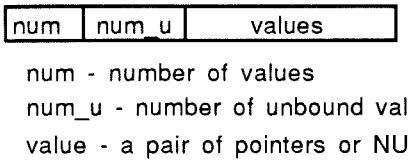
All the refinements have been implemented on a simulator [19] of the OR-Parallel token machine written in Simula67, running on a VAX750.

### 4.1 Straightforward

The structure of a directory is:




and of a context:




Initially a directory consisting of N (N=10) entries initialized to NULL is created for the root process. When a process creates children processes, each process, except one, gets a new directory as described in section 2, the last child inherits the parent's directory. When a new context is created all its values are initialized to NULL and its address is placed in the next free directory entry, possibly after a new directory has been created. If there are no free entries, a new directory with the size equal to the size of the current directory plus N is created, the entries in the current directories are copied to the new directory, and the new directory becomes current. Variables are accessed as in the basic storage model (section 2).

We propose two copying strategies: delayed copying of contexts, and copying of contexts on read. The delayed copying is used to avoid making local copies of contexts which are never used, and copying on read to increase locality. Delayed copying means that instead of copying an uncommitted context when new directories are created, a reference to the current copy is placed in all created directories, and the new copies are done when, and if, the content of the context is to be changed. Copying on read means that we make a local copy of a context not only on write but also on read. The distinction between committed and uncommitted context is relevant for delayed copying, but not for copy-on-read. Instead it must be known if a context is local (i.e. has been copied) to the local directory or not. The meaning of the num_u field in contexts is changed to indicate locality.We use all combinations of delayed copying/no delayed copying with copying on read/no copying on read. For each combination we count the total number of storage accesses, the numbers of local and remote accesses, and the number of initializations.

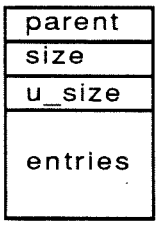
The following accesses are considered local: an initialization, an access to the current directory of a process, and an access to a context created in, or copied to, the current directory. This definition of locality, and also the definitions for the following implementations, are applicable to a

myriaprocessor with one process per processor. It is a reasonable approximation, if there are many processors, and the load is equally distributed among the processors. The definition is pessimistic, because in practice there will be more than one process per processing element.

### 4.2 Directory Trees

The main source of inefficiency in the straightforward implementation is the overhead for creating new directories. We propose the following remedy. When a process is created, an empty directory with a pointer to the parent directory is created, and the contexts in the ancestor directories are fetched to the current directory when needed. In the directory trees implementation we make a tradeoff between faster creation of processes and more complex variable access. This implementation model has been first specified in [9].

The structure of a context is unchanged. The structure of a directory is modified by adding the parent field:




parent - pointer to the parent directory
size - number of entries
u_size - number of used entries

entry - address to a context or NULL

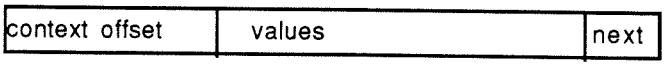
Some of the operations have been modified. When a child process is created, a directory with the same size as that of the current one is allocated, gets the address to the parent's directory and becomes current for the child process. Differently than in the previous implementation, the last, or the only, child does not inherit the parent's directory, because it is shared by all children. The new directory is empty, i.e. all its entries are initialized to NULL. When a new context is created it is initialized and its address is placed in the current directory. The access to variables becomes more complex. In order to read or write the value of a variable, we have first to find out if the context of the variable is directly accessible from the current directory. If it is, the value can be accessed in the usual manner. Otherwise the tree of directories must be searched. The search begins with the parent, and proceeds until the context is found. On a read operation the address of the accessed context is placed in the current directory, and on a write operation a local copy of the context is done.

As before we use the copying on read strategy. We also propose that when the directory for a child process is allocated, it gets the reference of the most recently created context in the parent's directory. We call this strategy the local context strategy. It is a meaningful strategy, because the contexts for the current clause and the clause invoking it are nearly always necessary for unification, and should be easily accessible. As before we use all combinations of the strategies, and collect the same data, plus the number of directory initializations.

### 4.3 Hashing on Contexts

In the directory tree implementation we reserve room in each directory for the references to all contexts in the binding environment of a process. It does not only make the process creation time consuming, but is also unnecessary because not all contexts are used. We propose using directories with a fixed size together with hash addressing of contexts. This time, we trade initialization time and memory space against more complex addressing. As a side effect of the decreased memory consumption, the total efficiency of the system will increase, because less time will be used for the garbage collection.

The u_size field is no longer needed in directories. Directories have room for a fixed number of entries, we call that part of a directory the hash window. Each entry points to a list of one or more contexts, a collision list. The structure of contexts is changed. Fields are added for identifying contexts and for resolution of collisions:




context_offset - id of the context
value - a pair of pointers or NULL
next - pointer to the next context in a collision list or NULL

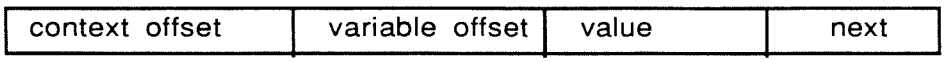
The address to the directory entry of a context equals: directory address + (context offset modulo size of the hash window). To access the value of a variable, first the collision list for the context of the variable is inspected, and in case the context is not found, the directory tree is searched. The other operations are the same as in the directory trees implementation, with the modification for hash addressing. As in the previous implementation, the directories are not inherited. That means that a hash window is created each time a clause is invoked, even if it is the only clause in a predicate. That has an interesting side-effect: the windows will not become overcrowded on recursive invocations of deterministic predicates.This scheme could be improved by creating a new window not on every invocation of a deterministic predicate, but only if the current window is too full.

As before we use all combinations of copying on read strategy and local contexts strategy. The combinations are investigated for the hash windows with sizes 2, 4, and 8. When the local contexts are used they are not accessed by hashing, we think it is a mistake to do so, because it increases the addressing overhead for all contexts. We collect the same type of data as before, plus the number of hashed accesses to the local and non-local hash windows, and the number of collisions. A collision occurs when the looked up context is not first in its collision list.

### 4.4 Hashing on Variables

The last optimization is based on the observation that not all variables in each context are used by each process. We propose a scheme where the values of variables are stored directly in directories, there are no contexts, and the hash addressing is used to access variables. This time we trade the copying time of contexts for even more complex variable access.

The structure of a directory is unchanged, but the content of entries is different. An entry contains a value frame consisting of the value of a variable and the fields needed for identifying the variable and for resolution of collisions:




<context offset, variable offset> - id of a variable
value - a pair of pointers or NULL
next - pointer to the next frame in a collision list or NULL

The value frame in a directory entry is the head of a collision list consisting of one or more value frames.

The address of a variable equals directory address + ((context offset*C + variable offset) modulo size of the hash window), where C is a prime number. To access a variable, first its collision list is inspected, and in case its value is not found the directory tree is searched. Notice that the tree need only be searched in case of read accesses, on write access the value is simply added to the collision list for the variable. When a clause is invoked no context is created, but instead value frames for the variables in the clause are placed in the local hash window with values initialized to NULL.

We use the same strategies as before, namely all combinations of local context and copying on read, with the following modifications: copying on read means copying a variable frame, and window sizes are 4, 8, and 16. When the local context strategy is used a context is created in the current directory on a clause invocation, and when a child process is created it gets the reference to the most recently created context in the parent's directory. The values in the local contexts are not accessed by hashing. For each access it is first decided if the value is in the local context or not. If it is, it is looked up directly. Otherwise the hashing procedure described above is applied. The same type of data is collected as in the previous implementation. When we count the number of directory initializations, we assume that it is enough to initialize one word per directory entry.

## 5. Programs

In order to investigate performance of the described implementations we use a very small benchmark consisting of three programs: permute, map, and queens. The programs has been chosen from the set of about 20 programs used in an investigation of the dynamic program behavior

[7]. We could neither use larger programs nor more programs, because of the prohibitive execution times of our simulator. Notice that the names starting with a lower case letter denote variables and predicates, while names starting with an upper case letter denote constants and data structures.

Permute:

```
p() <- perm(ulist,olist).
perm([],[]).
perm(x,[u|v]) <- del(u,x,z),perm(z,v).
del(x,[x|y],y).
del(x,[y|z],[y|w]) <- del(x,z,w).
```

is run with the list ulist consisting of 3, 4, 5, and 6 elements for all implementations and strategies, and of 8 for some.

Map:

```
/*
The program defines the relation color(Map,Colors),
between a map and a list of colors, which is true if Map is legally colored using Colors.
*/

p() <- test(map1).
test(map1) <- map(map1), colors(colorslst), color_map(map1,colorslst).

map( [Country(A,a,[b,c,d]),
      Country(B,b,[a,c,e]),
      Country(C,c,[a,b,d,e,f]),
      Country(D,d,[a,c,f]),
      Country(E,e,[b,c,f]),
      Country(F,f,[c,d,e]) ])
/*two, one or no countries are colored initially*/
  <- a = Red, b = Blue.

/* here is the program proper */

color_map([],_).
color_map([country|map],colorslst) <-
      color_country(country,colorslst),
      color_map(map,colorslst).

color_country(Country(_,c,adjacentCs),colorslst) <-
      remove(c,colorslst,colorslst1),
      subset(adjacentCs,colorslst1).

/* and here some utilities*/

subset([],_).
subset([c|cs],colorslst) <-
      remove(c,colorslst,_),
      subset(cs,colorslst).

remove(c,[c|cs],cs).
remove(c,[c1|cs],[c1|cs1]) <-
      remove(c,cs,cs1).

colors([Red,Green,Blue,White]).
```

is run with two, one or no countries colored from the beginning. The program is due to Ehud Shapiro.

Queens:

```
query() <- queens(conf).
queens(conf) <- goodboard(conf).

goodboard([]).
goodboard([x|y]) <- goodboard(y), nocollision(x,y).

nocollision(x,[]).
nocollision(x,[y1|y2]) <- notoppose(x,y1), nocollision(x,y2).

notoppose(P(x1,y1), P(x2,y2)) <- gen(y1), gen(y2),
      y1 /= y2, y1 - y2 /= x1 - x2, y1 - y2 /= x2 - x1.

gen(1).
gen(2).
gen(3).
gen(4).
/*facts below are used when there are more than 4 queens*/
gen(5).
/*facts below are used when there are more than 5 queens*/
gen(6).
 /*facts below are used when there are more than 6 queens*/
gen(7).
gen(8).
```

is run with the list conf consisting of 4, 5, and 6 elements for all implementations and strategies, and of 8 elements for some. An element in the list conf denotes the position of a queen. Calculating the depth for N-Queens (Table XII) we were also counting the calls to "/=".

## 6. Performance Evaluation

We first evaluate each implementation separately, and then compare them with each other. We show how the different strategies influence the speed and how locality pays off in myriaprocessor systems. Because of lack of space we do not show the complete results for all examples. They are presented in [19]. Performance for the two types of architectures is defined in section 3.

In the tables below, we use the following abbreviations. "delay" stands for "delayed copying of contexts", "copy" stands for "copying on read", and "local context" stands for "local contexts used".

### 6.1 Straightforward

In the dance hall type of systems there are no local memories, and the best performance is achieved when the total number of memory accesses is minimized, it is for strategies requiring the least copying. For all the examples the combination of delayed copying with no copying on read

performs best, and no delayed copying with copying on read worst. The ratio between the best and the worst grows with the size of the input structure up to 6. Table I shows the ratios between the total numbers of storage accesses for each combination of strategies (relative performance). The number of storage accesses is of course different for other examples, but the ratios are similar.

TABLE I
PERFORMANCE OF THE DANCE HALL, QUEENS

| number of queens | 4 | 5 | 6 |
|---|---|---|---|
| no delay, no copy | 1.00 | 1.00 | 1.00 |
| no delay, copy | 2.63 | 3.22 | 3.75 |
| delay, no copy | 0.65 | 0.62 | 0.62 |
| delay, copy | 1.07 | 1.02 | 0.96 |

For the myriaprocessor organization, the relations between the figures for the combinations of the strategies are similar. With the chosen relation between access times to local and remote memories the execution times for all implementations are shorter than on the dance hall organization. The improvements are proportional to the locality. Table II shows the ratios between the execution times for the different combinations, the locality, and the execution times relative to the times on the dance hall organization, for the queens program with 6 queens.

TABLE II
PERFORMANCE OF THE MYRIAPROCESSOR, 6 QUEENS

| | ratio | locality | time |
|---|---|---|---|
| no delay, no copy | 1.00 | 0.39 | 0.91 |
| no delay, copy | 3.33 | 0.50 | 0.80 |
| delay, no copy | 0.43 | 0.66 | 0.64 |
| delay, copy | 0.70 | 0.63 | 0.66 |

The rest of the examples show somewhat better locality and performance improvement.

### 6.2 Directory Trees

In this implementation the good locality becomes important even on the dance hall organization, because having local access to contexts decreases the search overhead. For the queens and permute programs the best performance is shown by the combination of the local context strategy and no copying on read strategy. For the map program the best is the combination of the local contexts and copying on read. The differences between the combinations of the strategies are quite small. It can be explained by the fact that though making a local copy of a context decreases the time for the following accesses, it costs extra storage accesses to copy a context. Table III shows the relative performance (the ratios between the total numbers of storage accesses for each combination of strategies) on the dance hall organization for the queens program.

TABLE III
PERFORMANCE ON THE DANCE HALL, QUEENS

| number of queens | 4 | 5 | 6 |
|---|---|---|---|
| no local context, no copy | 1.00 | 1.00 | 1.00 |
| no local context, copy | 1.49 | 1.44 | 1.37 |
| local context, no copy | 0.91 | 0.92 | 0.93 |
| local context, copy | 1.17 | 1.13 | 1.08 |

An interesting figure in this implementation is the high number of storage accesses used for initialization of directories. The number varies between 20% and 30% of all accesses. It can be compared to, approximately, 1% of accesses used for initialization of contexts. It also means that we can improve performance significantly by reducing the number of directory initializations.

For the myriaprocessor organization locality becomes even more important because it takes less time to access a local memory. Table IV shows the performance on the myriaprocessor organization for 6 queens.

TABLE IV
PERFORMANCE ON THE MYRIAPROCESSOR, 6 QUEENS

| | ratio | locality | time |
|---|---|---|---|
| no local context, no copy | 1.00 | 0.53 | 0.77 |
| no local context, copy | 1.17 | 0.65 | 0.65 |
| local context, no copy | 0.84 | 0.61 | 0.69 |
| local context, copy | 0.84 | 0.70 | 0.60 |

The figures for the other examples are similar.

### 6.3 Hashing on Contexts

In this implementation we introduce hash windows and try all combinations of local context strategy and copying on read strategy for the window sizes: 2, 4, and 8.

Performance improves slowly with the growing size of windows, but only for some combinations of strategies. For small windows (2) it is favorable to have local contexts and use no copying on read. It can be explained by the fact that the presence of local contexts decreases the number of hashed accesses and the number of collisions, and also using copying on read decreases number of collisions. For large windows (8), where the number of collisions is smaller, it is not even favorable to have local contexts. Table V shows the relative performance and the percent of collisions (in parenthesis) for the queens program, and Table VI the hit ratio, or the number of hashed accesses to the local contexts.

TABLE V
PERFORMANCE AND COLLISIONS ON THE DANCE HALL, QUEENS

| number of queens | 4 | 5 | 6 |
|---|---|---|---|
| hash window 2: | | | |
| no local context, no copy | 1.00 (25) | 1.00 (31) | 1.00 (36) |
| no local context, copy | 1.46 (40) | 1.39 (45) | 1.32 (49) |
| local context, no copy | 0.99 (4) | 1.00 (7) | 1.00 (3) |
| local context, copy | 1.22 (36) | 1.14 (41) | 1.07 (43) |
| hash window 4: | | | |
| no local context, no copy | 0.95 (10) | 0.94 (18) | 0.93 (18) |
| no local context, copy | 1.42 (27) | 1.35 (35) | 1.28 (38) |
| local context, no copy | 1.01 (0) | 1.02 (7) | 1.02 (3) |
| local context, copy | 1.23 (24) | 1.15 (37) | 1.08 (38) |
| hash window 8: | | | |
| no local context, no copy | 0.99 (6) | 0.95 (10) | 0.92 (8) |
| no local context, copy | 1.46 (22) | 1.37 (28) | 1.29 (29) |
| local context, no copy | 1.07 (0) | 1.07 (7) | 1.06 (1) |
| local context, copy | 1.29 (22) | 1.19 (33) | 1.11 (28) |

TABLE VI
HIT RATIO, QUEENS

| number of queens | 4 | 5 | 6 |
|---|---|---|---|
| no local context, no copy | 0.38 | 0.28 | 0.22 |
| no local context, copy | 0.63 | 0.57 | 0.52 |
| local context, no copy | 0.20 | 0.13 | 0.10 |
| local context, copy | 0.47 | 0.40 | 0.34 |

Another interesting figure is the number of initializations, this number grows with the size of windows from about 3% to 10% of all accesses. Even in the worst case it is only 20% of the number of initializations in the previous model.

We shall now take a closer look at how the choice of a strategy changes the performance. When a hash window grows, the number of collisions decreases and the number of initializations increases. When copying on read is used the hit ratio is good, but the number of collisions is high. When local contexts are used the hit ratio is very bad, but there are very few collisions. On the other hand the total number of hashed accesses decreases, because many variable accesses go to the local contexts. The different factors have counteracting influence on the performance, and this is why the differences between the different combinations of strategies are quite small.

For the myriaprocessor organizations the factors increasing locality become more important and outweigh losses caused by additional copying and conflicts. Table VII shows the performance on the myriaprocessor organization, locality, and the execution times relative to the times on the dance hall organization. The figures are similar for the other examples, though having local contexts is not as advantageous. It is worth noticing that very bad locality gives poorer performance on the myriaprocessor organization than on the dance hall organization.

TABLE VII. PERFORMANCE ON THE MYRIAPROCESSOR, 6 QUEENS

| | ratio | locality | time |
|---|---|---|---|
| hash window 2: | | | |
| no local context, no copy | 1.00 | 0.29 | 1.01 |
| no local context, copy | 1.01 | 0.53 | 0.77 |
| local context, no copy | 0.90 | 0.38 | 0.91 |
| local context, copy | 0.77 | 0.57 | 0.73 |
| hash window 4: | | | |
| no local context, no copy | 0.89 | 0.33 | 0.97 |
| no local context, copy | 0.95 | 0.55 | 0.75 |
| local context, no copy | 0.90 | 0.40 | 0.90 |
| local context, copy | 0.77 | 0.58 | 0.71 |
| hash window 8: | | | |
| no local context, no copy | 0.83 | 0.38 | 0.91 |
| no local context, copy | 0.91 | 0.58 | 0.71 |
| local context, no copy | 0.91 | 0.43 | 0.87 |
| local context, copy | 0.76 | 0.61 | 0.69 |

### 6.4 Hashing on Variables

In this implementation we try all combinations of local context strategy and copying on read strategy for different sizes of hash windows. This time the windows contain values of variables, and for this reason they are larger than before. The window sizes are: 4, 8, and 16.

The performance in this implementation is nearly independent of the window size, because when the window size grows the number of collisions decreases, but the number of initializations increases. Having local contexts is good, because it decreases the number of hashed variable access. It is more important than in the previous implementation, because hashed accesses to variables require more storage accesses than hashed accesses to contexts. Copying on read is advantageous, except for permute. Table VIII shows the relative performance and the percent of collisions (in parenthesis) for the queens program, and Table IX number of hashed accesses to the local variables.

TABLE VIII
PERFORMANCE AND COLLISIONS ON THE DANCE HALL, QUEENS

| number of queens | 4 | 5 | 6 |
|---|---|---|---|
| hash window 4: | | | |
| no local context, no copy | 1.00 (44) | 1.00 (47) | 1.00 (49) |
| no local context, copy | 1.10 (59) | 0.99 (63) | 0.91 (66) |
| local context, no copy | 0.69 (5) | 0.66 (2) | 0.65 (2) |
| local context, copy | 0.73 (38) | 0.65 (39) | 0.58 (44) |
| hash window 8: | | | |
| no local context, no copy | 0.87 (31) | 0.84 (33) | 0.82 (34) |
| no local context, copy | 0.96 (49) | 0.84 (52) | 0.75 (54) |
| local context, no copy | 0.72 (0) | 0.69 (1) | 0.68 (2) |
| local context, copy | 0.75 (29) | 0.65 (30) | 0.58 (33) |
| hash window 16: | | | |
| no local context, no copy | 0.87 (21) | 0.80 (21) | 0.76 (21) |
| no local context, copy | 0.94 (39) | 0.80 (39) | 0.70 (41) |
| local context, no copy | 0.80 (0) | 0.76 (1) | 0.73 (0) |
| local context, copy | 0.81 (22) | 0.70 (21) | 0.62 (22) |

TABLE IX
HIT RATIO, QUEENS

| number of queens | 4 | 5 | 6 |
|---|---|---|---|
| no local context, no copy | 0.35 | 0.27 | 0.22 |
| no local context, copy | 0.52 | 0.48 | 0.45 |
| local context, no copy | 0.19 | 0.12 | 0.09 |
| local context, copy | 0.39 | 0.33 | 0.28 |

Because of the larger window sizes the maximal number of initializations is doubled compared to the previous implementation, and goes up to 20% of all storage accesses.

For the myriaprocessor organization the picture of performance changes in the similar way as before, and the locality gets more important. Besides, having larger windows becomes more advantageous. It can be explained by the fact that initializations, which are always local, take relatively less time. Table X shows the performance figures on the myriaprocessor organization, the locality, and the execution times relative to the times on the dance hall organization. The figures are similar for the other examples. What the table does not show is that for all programs we use, the locality decreases slowly with the growing size of the input structure.

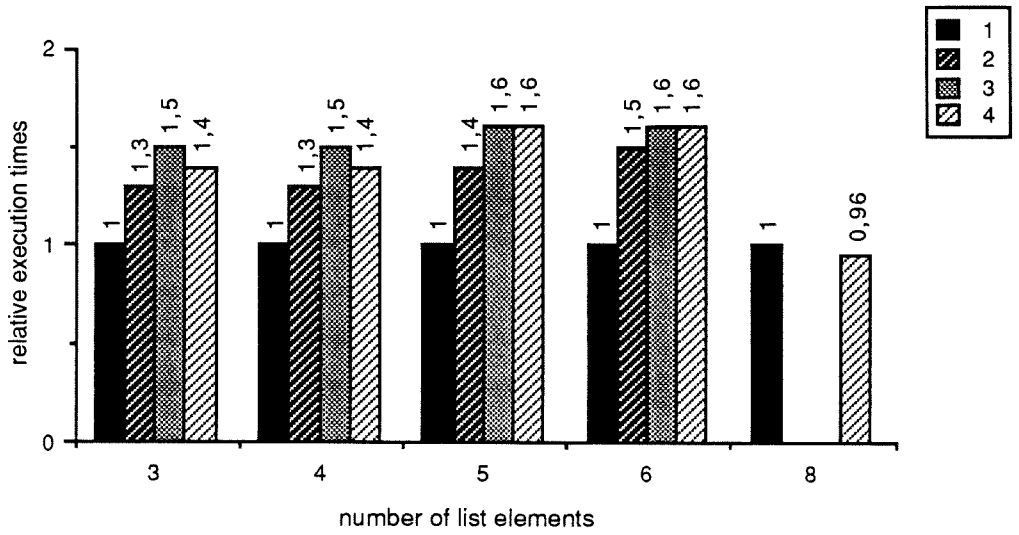
TABLE X
PERFORMANCE ON THE MYRIAPROCESSOR, 6 QUEENS

| | ratio | locality | time |
|---|---|---|---|
| hash window 4: | | | |
| no local context, no copy | 1.00 | 0.22 | 1.08 |
| no local context, copy | 0.75 | 0.41 | 0.89 |
| local context, no copy | 0.56 | 0.37 | 0.93 |
| local context, copy | 0.42 | 0.52 | 0.78 |
| hash window 8: | | | |
| no local context, no copy | 0.77 | 0.28 | 1.02 |
| no local context, copy | 0.56 | 0.49 | 0.81 |
| local context, no copy | 0.56 | 0.40 | 0.90 |
| local context, copy | 0.40 | 0.57 | 0.73 |
| hash window 16: | | | |
| no local context, no copy | 0.65 | 0.38 | 0.92 |
| no local context, copy | 0.46 | 0.59 | 0.71 |
| local context, no copy | 0.57 | 0.45 | 0.85 |
| local context, copy | 0.38 | 0.63 | 0.66 |

### 6.5 Comparison of Implementations

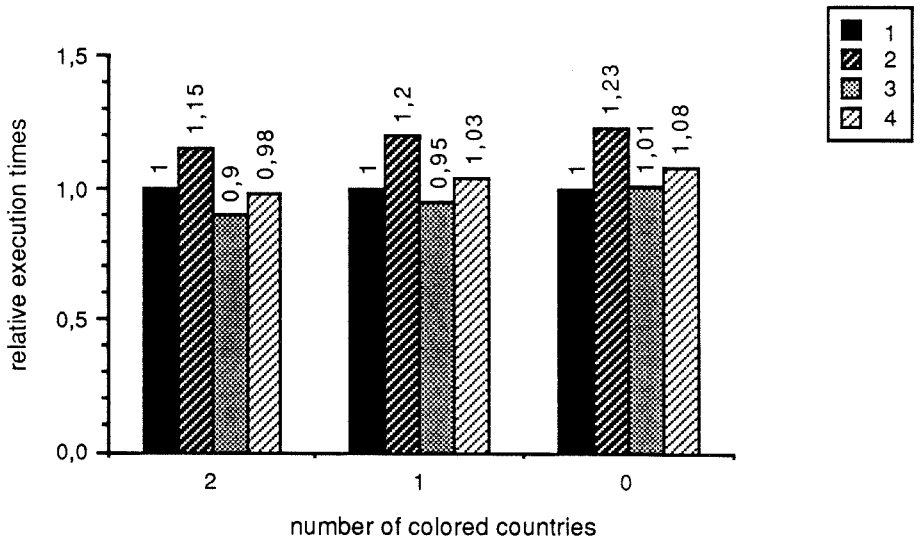
The performance of an implementation depends strongly on the executed program. What surprised us, was the very good performance of the straight forward implementation with delayed copying of contexts and no copying on read, especially for the permute program. Also worth noticing is the poor performance of the directory trees implementation. Figures 3, 4 and 5 show the relative performance on the dance hall organization for the best combination of strategies for each

implementation. The results are incomplete because of the prohibitive simulation times for the permute program for a list of length 8, and for the queens program for 8 queens.



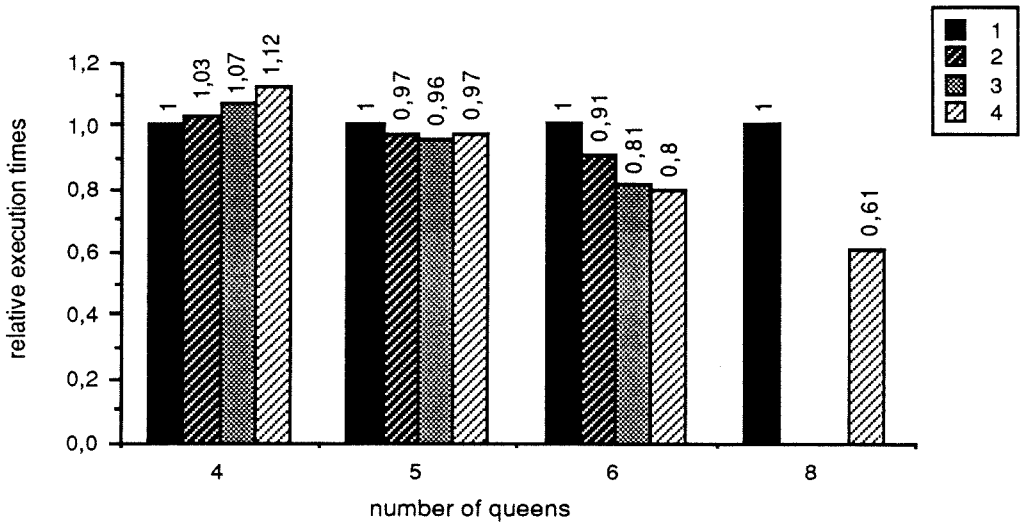

1 - straight forward, delay, no copy
2 - directory tree, local, no copy
3 - hashing on contexts, no local, no copy, hash window 8
4 - hashing on variables, local, no copy, hash window 4

FIGURE 3. COMPARISON OF IMPLEMENTATIONS ON THE DANCE HALL, PERMUTE




1 - straight forward, delay, copy
2 - directory tree, local context, copy
3 - hashing on contexts , no local context, copy, hash window 8
4 - hashing on variables, local context, no copy, hash window 16

FIGURE 4. COMPARISON OF IMPLEMENTATIONS ON THE DANCE HALL, MAP




1 - straight forward, delay, no copy
2 - directory tree, local context, no copy
3 - hashing on contexts , no local context, no copy, hash window 8
4 - hashing on variables, local context, copy, hash window 4

FIGURE 5. COMPARISON OF IMPLEMENTATIONS ON THE DANCE HALL, QUEENS

Implementations which perform well on the dance hall organization perform even better on the myriaprocessor organization. The improvement is proportional to the degree of locality. For the permute example the straight forward implementation is again superior, except for the list of length 8 when hashing on variables performs nearly as good. The tables comparing the performance of the implementations on a myriaprocessor are presented in [19].

It might seem surprising that the hashing implementations do not perform better in relation to the other ones. This fact can be explained by the design trade-offs. We have started with the straightforward implementation where a directory is scanned and plenty of contexts are copied on each process creation. We could improve the performance of this implementation by introducing delayed copying of contexts, without any penalty. Going over to the directory trees implementation the process creation is simplified by creating empty directories, but the directory entries must still be initialized. Besides, accesses to variables often require search of a directory tree. By introducing hashing on contexts we get constant overhead on process creation, because the directories (hash windows) have fixed size. Unfortunately using hashing makes access to variables even more complicated. Collision lists must be searched and contexts must be identified. Finally, introducing hashing on variables minimizes the amount of unnecessary copying, but the price is the need for variable identification, which is slightly more complicated than the identification of contexts. Table XI summarizes the important properties of each implementation.

TABLE XI
COMPARISON OF IMPLEMENTATIONS, TRADE-OFFS

| IMPLEMENTATION | STRENGTH | WEAKNESS |
|---|---|---|
| Straightforward | Simple variable access.<br>Selective copying of contexts. | Overhead on process creation grows with depth. |
| Directory tree | Simpler process creation.<br>Selective copying of contexts. | Overhead on process creation grows with depth.<br>Search of a directory tree on variable access. |
| Hashing on contexts | Constant overhead on process creation.<br>Selective copying of contexts. | Search of a collision list and a directory tree on variable access.<br>Complex context identification. |
| Hashing on variables | Constant overhead on process creation.<br>Selective copying of variables. | Search of a collision list and a directory tree on variable access.<br>Complex variable identification. |

To understand the difference in performance between the programs, and to be able to draw some more general conclusions, we have to know how the different properties of the programs influence performance. We could think about the following dynamic and static properties: depth of the search tree, degree of parallelism, distance (e.g. in number of calls) between branching points, size of contexts, number of contexts used between branching points, number of variables used in each context, number of times the instance of a variable is used. We shall discuss only two extreme implementations. The relative performance of the straightforward implementation should be good when the search tree is shallow (cheap process creation), distance between branching point is large (the same directory is used under several unifications), many variables in the same context are used (smaller copying overhead per variable), and finally when contexts are small (small copying overhead). On the other hand the implementation using hashing on variables should perform well compared to other implementations when the search tree is deep, there is much parallelism (many branching points), contexts are large, and finally, few values are used in each context and each value is used several times. The discussion could be summarized by saying that the more complex implementations will perform relatively better on more complex programs running on large data structures. Table XII shows values for some of the named factors. The content of the table and the previous data confirm that the performance is strongly dependent on the depth of the search tree, and also on the degree of parallelism. It does not say anything about the importance of the context size and the number of accesses to the same context, because the mean size of contexts and the mean number of accesses to the same context vary very little.

TABLE XII
PROPERTIES OF PROGRAMS

| | | DEPTH | DEGREE OF PARALLELISM | MEAN CONTEXT SIZE | MEAN NUMBER OF ACCESSES |
|---|---|---|---|---|---|
| Permute | 3 | 10 | 8 | 3 | 3 |
| | 4 | 14 | 24 | 3 | 3 |
| | 5 | 20 | 85 | 3 | 3 |
| | 6 | 27 | 381 | 3 | 3 |
| | 8 | 44 | 12800 | 3 | 3 |
| Map | 2 | 99 | 37 | 3 | 3 |
| | 1 | 104 | 106 | 3 | 3 |
| | 0 | 104 | 425 | 3 | 3 |
| Queens | 4 | 52 | 12 | 3 | 2 |
| | 5 | 81 | 48 | 3 | 2 |
| | 6 | 118 | 196 | 4 | 2 |
| | 8 | 213 | 3380 | 4 | 2 |

The presented comparison asks for some qualifications. Earlier in this paper we have named two doubtful choices we have made: mixed addressing when local contexts are used in the hashing on contexts implementation, and creation of a new directory (hash window) even on invocations of deterministic predicates in both hashing implementations. Especially changing the second decision, as suggested in Section 4.3, would change the results in favor of the hashing implementations. There is another, more important factor, which could change the results. In most sequential implementations of Prolog the indexing technique is used in order to avoid creation of unnecessary backtracking points and unifications. In parallel implementations indexing will be even more important because it will prevent relatively expensive creation of processes, which will surely terminate during the first unification. For example, in the permute program two processes are created for each invocation of the perm predicate, whereas if indexing were used only one would be created. Results from our earlier investigation [7] show that up to 90% of all branches can be eliminated by indexing. Introducing indexing would favor the straightforward implementation most, because of the high price of process creation.

There are many other improvements of technical and logical nature which could be incorporated in an implementation. We have kept our implementations as simple as possible, and have concentrated on the most important factors, in order to be able to interpret the results of the simulations.

## 7. Conclusions

We have presented simulation results for four implementations of the basic storage model for OR-Parallelism. The results show how the different factors influence performance of the implementations. On the basis of the obtained results we claim that the implementation using

hashing on variables would perform best for the "real life" large programs. Our results confirm findings of Crammond [8] who has investigated a hashing implementation which is quite similar to ours.

Performance figures for the myriaprocessor organization show (under our assumptions) that a system of this type would run about 20-30% faster than a system of the dance hall type, with most of the implementations. Those figures could be used to predict performance of a system with a global storage and caches. In such a system the data classified as local (directories and some contexts) could be held in caches. This would favor the hashing implementations, where the sets of local data, defined as the content of the local hash windows, are quite small. Recall that the size of the largest hash windows is 16, and even if the "real life" programs would require larger windows, still several windows for each branch could be stored in a cache.

We have assumed that a processor has no support for hashing (for accessing the hash addressed objects in the memory), except for computing of hash functions. That means that, in the best case, hashed access to a variable would require two storage accesses more than an ordinary access. Our assumption is very pessimistic, because a processor could be equipped with a better support for hashed accesses (for instance overlapping between identifying elements and searching the collision list).

We expect that on systems with a global storage, caches, and more support for hashed accesses the hashing implementations (especially hashing on variables) would beat the other implementations much more clearly than in our simulation.

The work presented in this paper, and also other reported simulations of computational models and computer architectures [8, 20, 21, 22] raise the question of the usefulness of the simulation technique as a tool in developing new computer architectures. Can we really make architectural choices on the basis of simulation results? The reasons for our doubts are the limited size and number of the programs, which can be executed on the simulated machines, the level of detail of the simulations, and also some of the assumptions, e.g. no memory and bus contention. Our answer is that simulations can not give the ultimate answers to the design questions, but if only the judiciously chosen parts of the machine are simulated on a detailed level, then the obtained results can give a very good guidance in making design choices. Another question open to debate is whether an OR-parallel machine buys anything in terms of performance and complexity over a purely sequential machine. The overhead and added complexity of exploiting OR-parallelism are substantial.

Our next step towards the computer architecture for parallel execution of logic programs is the implementation of an OR-Parallel Prolog system on one of the several commercial multiprocessor systems with shared memory, which have appeared on the market lately, for example Multimax [12], Balance [13], or Butterfly [15]. Our Prolog system will contain most of the built-in functions found in the sequential implementations of Prolog, and also facilities for controlling parallelism, which would replace ordering of clauses and the cut primitive used in ordinary Prolog. The facilities for controlling execution will be even more important in parallel systems, than they are in sequential ones.

**Acknowledgements**

Most of this work was done while the authors were employed at the Department of Computer Systems of the Royal Institute of Technology. We thank everyone at the department, and in particular professor Lars-Erik Thorelli, for making this work possible. We also thank our colleagues at SICS, Thomas Sjöland, Lennart Fahlén, Dan Sahlin, and Seif Haridi for many useful comments on the contents of this paper.

**References**

[1] M.G.H. Katevenis, Reduced Instruction Set Computer Architecture for VLSI, PhD Thesis, The MIT Press 1985.

[2] P.F.Wilk, Prolog Benchmarking, D.A.I. Research Report , Edinburgh University, 1984.

[3] D.H.D.Warren, Implementing Prolog - Compiling Predicate Logic Programs, vol 2, D.A.I. Research Report no 40, Edinburgh University, 1977.

[4] L.Raskin, Performance Evaluation of Multiple Processor Systems, PhD Thesis, CMU, Computer Science Tech. Report CMU-CS-78-141, 1978.

[5] A.K.Jones, E.F.Gehringer, eds., The Cm* Multiprocessor Project: A Research Overview, CMU, Computer Science Tech. Report CMU-CS-80-131, 1980.

[6] D.H.D.Warren, An Abstract Prolog Instruction Set, SRI International, Technical Note 309, 1983.

[7] A.Ciepielewski, B.Hausman, S.Haridi, Initial Evaluation of a Virtual Machine for OR-Parallel Execution of Logic Programs, in IFIP TC-10 Working Conference on Fifth Generation Computer Architecture, UMIST Manchester, 1985.

[8] J.Crammond, A Comparative Study of Unification Algorithms for OR-Parallel Execution of Logic Languages, IEEE Transaction on Computers, vol c-34, no 10, October 1985.

[9] A.Ciepielewski, S.Haridi, A Formal Model for Or-parallel execution of Logic Programs, in IFIP 83, North Holland P. C., Mason (ed).

[10] R.H.Halstead, Jr., Processor Architecture for Multiprocessors, MIT Laboratory for Computer Science, January 1985.

[11] J.Schwartz, "Ultracomputers", ACM Transactions on Programming Languages and Systems, vol 2, no 4, October 1980.

[12] Multimax Technical Summary, Encore Computer Corporation, 1986.

[13] Balance 8000, System Technical Summary, Sequent Computer Systems, Inc., 1984.

[14] G.F.Pfister, et al., The IBM Research Parallel Processor Prototype: Introduction and Architecture, in Proceedings of the 1985 Intl. Conf. on Parallel Processing.

[15] BBN, "Development of a Voice Funnel System: Quarterly Technical Report", BBN Reports #4845 (Jan. 1982) and #5284 (April 1983), Bolt, Beranek, and Newman, Cambridge, Mass.

[16] MC68020 32-Bit Microprocessor, User Manual, Motorola Inc. .

[17] VMEbus Specification Manual, VMEbus Manufacturers Group.

[18] R.H.Halstead, Jr., et al., Concert: Design of a Multiprocessor Development System, MIT Laboratory for Computer Science, January 1985.

[19] B.Hausman, Empirical Studies of OR-Parallel Execution of Logic Programs, License report, TRITA-CS-8602, April 1986, Dept. of Computer Systems, The Royal Institute of Technology.

[20] K.P.Gostelow, R.E.Thomas, Performance of a Simulated Dataflow Computer, IEEE Transaction on Computers, vol c-29, no 10, October 1980.

[20] R.Onai, et al., Architecture of a Reduction-Based Parallel Inference Machine: PIM-R, New Generation Computing, 3 (1985).

[21] N.Ito, et al., The Architecture and Preliminary Evaluation Results of the Experimental Parallel Inference Machine PIM-D, ICOT, 1986.

[22] P.Borgwardt, Parallel Prolog using stack segments on shared-memory multiprocessors, in Proc. 1984 Int. Symp. Logic Programming, Feb. 1984, pp. 2-11.