DATA-PARALLEL CONCURRENT CONSTRAINT PROGRAMMING

By

Bo-Ming Tong

A Thesis

submitted in partial fulfillment of the requirements

for the Degree of Master of Philosophy

Division of Computer Science

Graduate School

The Chinese University of Hong Hong

June 1994

Acknowledgement

I wish to thank my supervisor, Dr Ho-Fung Leung, without whose continuous support and encouragement this thesis would not have been possible. At times I was arrogant and over-confident with my own abilities, yet at other times I was ignorant of even the most basic concepts in logic programming. Yet at all times, Dr Leung guided me along with patience. Gradually, I learned how to do research. We have always worked together closely, and his ideas were always sound and insightful. For instance, he first suggested to me that Parlog could be implemented on a data-parallel computer and that finite domain constraints could be incorporated into a committed-choice language. This thesis turns out to be a realization of these key concepts.

I wish to thank also Dr Jimmy Ho-Man Lee, who taught me a great deal on logic programming and on technical writing. Dr Kam-Wing Ng introduced me to the exciting realm of logic programming and guided me through in the final year project of my undergraduate studies.

Tak-Wai Lee devised the trailing scheme of suspension list updates described in this thesis and all credits should go to him. Chong-Kan Chiu informed me that the idempotence property of interval narrowing does not hold for finite domains in general. I must also thank Daniel Diaz and Donald A. Smith, for

numerous insightful email discussions, and all the anonymous referees of my papers, who gave many useful comments which lead to the improvement of this thesis.

I would like to thank the system administrators, Philip Chan and Angus Siu, for helping me whenever I encountered any problems concerning the DECmpp massively parallel computer. Larry Levine of MasPar Customer Support helped me in obtaining benchmark results.

Advices from the LaTeX guru Chung-Yuen Li was the most useful whenever I encountered problems in the preparation of this manuscript. Chi-Shing Leung taught me how to use the PiCTeXsystem and provided me with lots of examples.

To my grandmother

Abstract

With the advent of cost-effective massively parallel computers, researchers conjecture that the future concurrent constraint programming system is composed of a massively parallel constraint solver as the back-end with a concurrent inference engine as the front-end [Coh90]. This thesis represents probably the first attempt to build a concurrent constraint programming system on a massively parallel SIMD computer. A concurrent constraint programming language called Firebird is presented. Firebird can handle finite domain constraints and supports both concurrency and data-parallelism. As a result, it is suitable for implementation on both multiprocessors and SIMD computers. Concurrency arises from the stream and-parallelism of committed-choice logic programming languages. In a *nondeterministic derivation step*, one of the domain variables is selected to create a choice point. All possible alternatives are attempted in parallel. Data-parallelism is exploited in the resulting or-parallel execution.

The *Data-Parallel Abstract Machine* (DPAM) has been designed as the basis of implementation of Firebird. The *concurrent process scheduler* uses simple data-parallel algorithms for process resumption and deadlock detection. On the other hand, when the number of processor elements is not enough for exploiting or-parallelism, the system resorts to *parallel backtracking* automatically. We

present the data structures necessary for maintaining a vector trail stack and methods for restoring all process queues to their original states upon backtracking.

An almost complete implementation of Firebird has been built on a DECmpp 12000 Sx-100[1] massively parallel computer with 8,192 processor elements. The performance of the implementation is compared with CHIP, a commercial sequential implementation. Our performance figures indicate that the parallelism attained is usually counterbalanced by the poor performance of individual processor elements. Nevertheless, our results also indicate that a speedup of 2 orders of magnitude is possible when we compare the performance using 8,192 processor elements and the performance using a single processor element of the same machine. Furthermore, the speedup is scalable, provided that the problem size is large enough for effective exploitation of or-parallelism. On the other hand, we measured the effects of several control strategies and optimizations on execution time and memory consumption in a data-parallel context.

[1]DECmpp 12000 Sx-100 is equivalent to MasPar MP-1.

Contents

1 Introduction **1**

1.1 Concurrent Constraint Programming 2

1.2 Finite Domain Constraints 3

2 The Firebird Language **5**

2.1 Finite Domain Constraints 6

2.2 The Firebird Computation Model 6

2.3 Miscellaneous Features 7

2.4 Clause-Based Nondeterminism 9

2.5 Programming Examples 10

2.5.1 Magic Series 10

2.5.2 Weak Queens 14

3 Operational Semantics **15**

3.1 The Firebird Computation Model 16

3.2 The Firebird Commit Law 17

3.3 Derivation 17

3.4 Correctness of Firebird Computation Model 18

4 Exploitation of Data-Parallelism in Firebird **24**

4.1 An Illustrative Example 25

4.2 Mapping Partitions to Processor Elements 26

4.3 Masks 27

4.4 Control Strategy 27

4.4.1 A Control Strategy Suitable for Linear Equations 28

5 Data-Parallel Abstract Machine **30**

5.1 Basic DPAM 31

5.1.1 Hardware Requirements 31

5.1.2 Procedure Calling Convention And Process Creation . . 32

5.1.3 Memory Model 34

5.1.4 Registers 41

5.1.5 Process Management 41

5.1.6 Unification 49

5.1.7 Variable Table 49

5.2 DPAM with Backtracking 50

5.2.1 Choice Point 52

5.2.2 Trailing 52

5.2.3 Recovering the Process Queues 57

6 Implementation **58**

6.1 The DECmpp Massively Parallel Computer 58

6.2 Implementation Overview 59

6.3 Constraints 60

6.3.1 Breaking Down Equality Constraints 61

6.3.2 Processing the Constraint 'As Is' 62
6.4 The Wide-Tag Architecture 63
6.5 Register Window 64
6.6 Dereferencing 65
6.7 Output 66
6.7.1 Collecting the Solutions 66
6.7.2 Decoding the solution 68

7 Performance **69**
7.1 Uniprocessor Performance 71
7.2 Solitary Mode 73
7.3 Bit Vectors of Domain Variables 75
7.4 Heap Consumption of the Heap Frame Scheme 77
7.5 Eager Nondeterministic Derivation vs Lazy Nondeterministic Derivation 78
7.6 Priority Scheduling 79
7.7 Execution Profile 80
7.8 Effect of the Number of Processor Elements on Performance . . 82
7.9 Change of the Degree of Parallelism During Execution 84

8 Related Work **88**
8.1 Vectorization of Prolog 89
8.2 Parallel Clause Matching 90
8.3 Parallel Interpreter 90
8.4 Bounded Quantifications 91
8.5 SIMD MultiLog 91

9 Conclusion **93**

9.1 Limitations 94

9.1.1 Data-Parallel Firebird is Specialized 94

9.1.2 Limitations of the Implementation Scheme 95

9.2 Future Work 95

9.2.1 Extending Firebird 95

9.2.2 Improvements Specific to DECmpp 99

9.2.3 Labeling 100

9.2.4 Parallel Domain Consistency 101

9.2.5 Branch and Bound Algorithm 102

9.2.6 Other Possible Future Work 102

Bibliography **104**

List of Tables

2.1 Informal rules for evaluation of cardinality constraints 12

5.1 Comparison of suspension list update with and without time stamp 54

7.1 Benchmark set 70

7.2 Benchmark: uniprocessor performance 72

7.3 Execution time of machine instructions (in machine cycles) . . . 72

7.4 Benchmark: solitary mode performance 74

7.5 Benchmark: on demand creation of bit vectors 76

7.6 Benchmark: heap fragmentation 77

7.7 Benchmark: lazy nondeterministic derivation vs eager nondeterministic derivation 78

7.8 Benchmark: priority scheduling 80

7.9 Benchmark: execution profile (in %) 81

List of Figures

4.1 Constraints remaining after first deadlock 25

4.2 Or-parallel branches in the 5-queens example 26

5.1 Example: mask bit vector 33

5.2 Memory areas and scalar registers 35

5.3 Example: tag-on-term and tag-on-data representations of `a(b,c)` 35

5.4 Example: a *h*-variable 36

5.5 Example: a *d*-variable 37

5.6 Example: building of a heap frame 40

5.7 Suspension 44

5.8 Example: resumption of suspension lists 46

6.1 Heap cell format 63

6.2 Examples: reference pointer, atom, list and compound term . . 64

6.3 Unbound variable representations of WAM and DPAM 64

6.4 Register windows 65

7.1 Benchmark: run time of n-queens program 82

7.2 Benchmark: speedup of N-Queens Program 83

7.3 Execution trace of 8-queens, $\#proc$=8,192 85

7.4 Execution trace of 8-queens, $\#proc$=64 87

Chapter 1

Introduction

With the advent of cost-effective massively parallel computers, researchers conjecture that the future concurrent constraint programming system is composed of a massively parallel constraint solver as the back-end with a concurrent inference engine as the front-end [Coh90]. This thesis represents probably the first attempt to build a concurrent constraint programming system on a massively parallel SIMD computer.

Shared-memory multiprocessors have been regarded as the architecture of choice in traditional concurrent constraint programming research. Efficient implementations with near linear speedup has been reported [Cra90a, CWY91] but the inherent bus contention bottleneck of this architecture makes massive parallelism impossible. Therefore, we have chosen one of the most scalable architectures instead—the SIMD architecture with distributed local memory.

A new finite domain constraint language called *Firebird* has been designed. Its syntax is similar to mainstream concurrent logic programming languages, and in particular, *flat GHC* [Ued85]. The most distinguishing semantic feature

of this language is that committed-choice indeterminism is integrated with don't know nondeterminism by introducing the notion of domain-variable-based choice points. In an *indeterministic derivation step*, execution consists of guard tests, commitment and spawning in the same manner as committed-choice languages. In a *nondeterministic derivation step*, one of the domain variables in the system is chosen and all possible values in its domain are attempted in an or-parallel manner. Alternatively, a choice point based on the domain variable is set up and each possible value in its domain is attempted by backtracking.

Firebird could be implemented efficiently on many architectures. On sequential machines and shared-memory multiprocessors it is expected to be more efficient than languages using the Andorra Model [War90] because no *determinacy test* is needed. On data-parallel computers, or-parallelism is exploited by attempting all possible values in the domain of a variable in parallel after the variable has been labeled. In this way, thousands of finite domain constraints can be solved in a single step.

In the rest of this chapter, we shall present the preliminaries. The Firebird language will be introduced in chapters 2 and 3. The data-parallel execution model, its implementation and evaluation will be presented in chapters 4–7. We cite related work in chapter 8. Finally, we conclude and give suggestions of future work in chapter 9.

1.1 Concurrent Constraint Programming

ALPS [Mah87] is a scheme to integrate constraint logic programming and concurrent logic programming. Saraswat [Sar88, SR90] developed the ideas further

by introducing the concurrent constraint programming framework. Computation is modeled as the interaction of concurrent, cooperating *agents*[1] exchanging information via a global *store*, which is a conjunction of *constraints*. An agent may assert (*tell*) new constraints to the store, as well as query (*ask*) whether a constraint is implied (*entail*) by the store. The constraints in the store must be consistent (*satisfiable*), or the computation *aborts*.

Since each tell constraint is conjoined to the current store, the store is *monotonically refined*. As a result, a successful ask operation will remain successful throughout the rest of the computation. Thus, synchronization can be achieved by *blocking ask*—an agent blocks until the store is refined enough to entail the constraint it wants to ask. It remains blocked until some other concurrently executing agents have added enough information to the store so that it is strong enough to entail the ask constraint. However, this may never happen. It is also possible that the ask constraint is simply *unsatisfiable*, and the computation aborts.

1.2 Finite Domain Constraints

A *domain* is a finite non-empty set of constants. A *domain variable*, or simply a d-*variable*, is a variable which ranges over a domain. Recent treatment of finite domain constraints in the concurrent constraint programming framework [VHSD93, DC93] represents a domain variable X with domain d as a constraint $X \in d$. As constraints are added to the store, the domain of each related variable shrinks, until it becomes a *singleton*. For example, X may take any value from 1

[1] An agent corresponds to a goal in traditional logic programming.

to 10 initially. A constraint $X > 4$ will rule out some of the elements in d. Now X can only range from 5 to 10. When a constraint $X < 6$ is added, X becomes a singleton, and X is equal to 5. Usually, a disequality constraint $X \neq Y$ will block until either X or Y is bound. If all constraints and goals block then a *deadlock* or a *floundering* is said to have occurred. To avoid deadlock there is usually a system predicate which attempts each possible value of a domain variable and the variable is said to be *labeled.*

The reader is referred to [VH89] for a full treatment of finite domain constraints in the traditional logic programming framework [Llo87]. It is summarized as follows. Ordinary variables are termed h-variables (h stands for *Herbrand*). A d-variable X with domain d is denoted by X^d. The unification algorithm must be modified to support d-variables. The modified algorithm is termed d-*unification.* When a h-variable is unified with a d-variable, the former is bound to the latter. When a constant c is unified with a d-variable X^d, X is bound to c if c is in d. Otherwise the unification fails. When two d-variables, X^d and Y^e, are unified, both of them are bound to the d-variable Z^f where $f = d \cap e$. If f is a singleton $\{c\}$, both variables are bound to the constant c. If f is empty, the unification fails. SLD resolution extended with d-unification is termed *SLDD resolution.* However, the introduction of d-unification alone is insufficient to solve finite domain constraints efficiently. Disequality, inequality, (arithmetic) equality constraints, and even user-defined constraints, must also be handled. The *forward checking* and *looking ahead* inference rules are introduced as both a theoretical basis and an implementation scheme for such constraints.

Chapter 2

The Firebird Language

The syntax of Firebird is almost identical to flat GHC [Ued85]. Each clause consists of a head, a guard part (consisting of ask constraints) and a body (tell constraints and goals). The following is a Firebird program which solves the *n-queens problem.*

```
queen(N,L) :- gen_list(N,L), L in 1..N, constraint(L).

constraint([]).
constraint([X|T]) :- safe(X,T,1), constraint(T).

safe(X,[],N).
safe(X,[Y|T],N) :- noattack(X,Y,N), N2 is N + 1, safe(X,T,N2).

noattack(X,Y,N) :- X #\= Y, X #\= Y + N, X #\= Y - N.

gen_list(0,L) :- L = [].
gen_list(N,L) :- N \== 0 | L = [H|T], N2 is N - 1, gen_list(N2,T).
```

2.1 Finite Domain Constraints

In Firebird programs, finite domains are denoted by $l..u$. The domain of an unbound variable X can be specified by `in/2`. A list of unbound variables can be initialized at once. For example,

```
[X] in 1..10.
[A,B,C] in 1..100.
```

In CHIP [DVHS+88], a domain declaration is used to specify the domain of an argument of a predicate. The argument will be unified with a new domain variable with the specified domain when the predicate is called. A domain declaration can be regarded as an implicit unification in the head matching phase. On the other hand, there are no domain declarations in Firebird. This is because the flat GHC rule that all arguments are in input mode and all unifications must be stated explicitly in the body will be violated otherwise.

Permitted ask and tell constraints include $=, \neq, <, \leq, >, \geq$ on any linear expressions. We extend the suspension rule of flat GHC to accommodate ask constraints. If an ask constraint attempts to reduce the domain of a global variable, it will also suspend. For example, if the domain of variable X is {1,7}, then the ask constraint $X > 6$ will suspend until some other goal remove 1 from the domain of X.

2.2 The Firebird Computation Model

The *Firebird Computation Model* is a new approach to handle finite domain constraints in concurrent constraint programming languages. Unlike the *Andorra*

Computation Model [War90] in which nondeterministic goals are the basis for setting up choice points or the exploitation of or-parallelism, the Firebird Computation Model uses domain variables instead. Intuitively, there are two kinds of derivation steps in Firebird, called the *indeterministic derivation step* and the *nondeterministic derivation step.* An indeterministic derivation step is identical to a derivation step in a committed-choice concurrent constraint programming language. In a nondeterministic derivation step, if there is an unbound domain variable X in the system with domain $\{a_1, \ldots, a_n\}$, Firebird will create n or-parallel branches, each of which executes with an additional constraint $X = a_i, 1 \leq i \leq n$. Alternatively, a choice point based on X can be set up and each of the values is attempted by backtracking. X is said to be *labeled.*

Execution consists of alternating indeterministic and nondeterministic derivations. How the two are interleaved is unspecified and left to the implementation. At the lazy nondeterminism extreme, indeterministic derivation always takes precedence, and consequently nondeterministic derivation is only used to resolve deadlocks. At the eager nondeterminism extreme, nondeterministic derivation always takes precedence. Our data-parallel implementation uses a control strategy which lies somewhere in between these two extremes.

2.3 Miscellaneous Features

If `::/2` is used in place of `in/2`, the *d*-variables created will never be subject to a nondeterministic derivation step.

If no processes (goals and constraints can be regarded as concurrent processes) suspend on a *d*-variable, the variable will not be used in a nondeterministic derivation because labeling the variable is futile

If X is originally a *h*-variable, but becomes a *d*-variable when its domain can be inferred from a constraint (*e.g.* $X = Y + 1, Y \in \{1,2\}$), X will never be subject to a nondeterministic derivation. It is inefficient to label both X and Y since they are related to each other. Applying nondeterministic derivation to Y alone is sufficient. Ideally, the entire constraint network can be analyzed to determine which *d*-variables should be used in nondeterministic derivations. However, if the programmer creates no more *d*-variables using `in/2` than necessary and let the system infer the domain of other *d*-variables, such expensive analysis becomes unnecessary.

In Firebird, it is not allowed to build arithmetic expressions at runtime and pass them to tell constraints like the CHIP system [DVHS+88]. We justify this omission by showing the semantic complications which may arise. Suppose there is a tell constraint $X = Y + Z$ where X is an unbound *h*-variable and Y, Z are unbound *d*-variables with domain 0..5. In Firebird, X is bound to a domain variable with domain 0..10 automatically. Had we allowed X to be bound to an arithmetic expression like $A + B$ at runtime, we could have been forced to use an unification algorithm which interpret arithmetic functions, resulting in a tell constraint $X = A + B$.

We find the ability to bind a *h*-variable to a *d*-variable automatically very important because sometimes initial domains of variables in a tell constraint need not be specified. To avoid the complications described above, runtime binding of arithmetic expressions to variables in a tell constraint is treated as

an error.

2.4 Clause-Based Nondeterminism

Firebird, though a committed-choice language, is by no means less expressive than a language adopting the Andorra model [War90]. This is because clause-based nondeterminism can be emulated using domain-variable-based nondeterminism. A naïve way of mechanically translating flat Pandora [Bag91] programs to Firebird is as follows. For each don't know flat Pandora procedure P

$$P \leftarrow G_{1,1}, \ldots, G_{1,m_1} \mid B_1$$
$$P \leftarrow G_{2,1}, \ldots, G_{2,m_2} \mid B_2$$
$$\vdots$$
$$P \leftarrow G_{n,1}, \ldots, G_{n,m_n} \mid B_n$$

where n is the number of clauses and m_i is the number of ask constraints in clause i, construct a Firebird clause

$$P \leftarrow x \in 1 \ldots n, A_1, \ldots, A_n$$

where x is a new domain variable which does not appear in P or any $G_{i,j}, B_i$. For each ask constraint $G_{i,j}, 1 \leq i \leq n, 1 \leq j \leq m_i$ add a clause

$$A_i \leftarrow \neg G_{i,j} \mid x \neq i$$

where $\neg G_{i,j}$ denotes the negation of $G_{i,j}$. Finally, add

$$A_i \leftarrow x = i \mid G_{i,1}, \ldots, G_{i,m_i}, B_i$$

for each i, $1 \leq i \leq n$. The idea is to use x to denote the set of candidate clauses. If any ask constraint in clause i fails, i will be removed from the domain of

x and clause i will never be considered a candidate clause. If only clause i is satisfiable, x will become a singleton, resulting in a commitment to clause i. The ask constraints will be told to the store as in ALPS [Mah87] and the Andorra Model [War90]. Finally, if a deadlock occurs, x will be labeled by the system using nondeterministic derivation and as a result all the satisfiable clauses will be attempted. With some adaptation, the decision graph compilation technique for nondeterminate concurrent logic programs [KT91] can be used in place of our naïve translation algorithm.

2.5 Programming Examples

2.5.1 Magic Series

The magic series $s_0, s_1, \ldots, s_n$ is a non-empty finite series of non-negative integers, such that $\forall i,\ 0 \leq i \leq n,\ s_i = |\{j : 0 \leq j \leq n,\ s_j = i\}|$, where the cardinality of a set S is denoted by $|S|$.

2.5.1.1 Using Choice Points

To solve this problem, we count, for each position i, the total number of s_j's which is equal to i. Since the s_j's are initially unknown, to count the number of occurrences of i, we could make choices, one assuming $s_j = i$ and the other assuming $s_j \neq i$. This approach is taken by [VH89] and the program is shown below.

```
magic(N,L) :-
        M is N + 1,
        length(L,M),
        L :: 0..M,
        occurrences(L,0,L),
        labeling(L).

occurrences([],N,L).
occurrences([X|Y],N,L) :-
        occur(X,N,L),
        N1 is N + 1,
        occurrences(Y,N1,L).

occur(0,Val,L) :-
        outof(Val,L).
occur(N,Val,[Val|Tail]) :-
        N > 0,
        N1 is N - 1,
        occur(N1,Val,Tail).
occur(N,Val,[First|Tail]) :-
        N > 0,
        Val #\= First,
        occur(N,Val,Tail).

outof(X,[]).
outof(X,[H|T]) :-
        X #\= H,
        outof(X,T).

labeling([]).
labeling([H|T]) :-
        indomain(H),
        labeling(T).
```

In Firebird, the clause-based nondeterminism of the above program can be emulated by using domain variables. Though much more efficient than using backtracking alone, the search space is not pruned in a completely "*a priori*" manner.

2.5.1.2 Using Cardinality Constraints

[HD91] tackles this problem by introducing the *cardinality operator* #. The cardinality constraint $\#(l, u, c_1, c_2, \ldots, c_n)$ states that the number of constraints c_i, $1 \leq i \leq n$, which are true, lies between l and u. Thus, we could evaluate the cardinality constraint by the following informally defined rules. The reader is referred to [HD91] for formal definitions.

Trivial Satisfaction	if $l \leq 0 \wedge u \geq n$, the cardinality constraint is trivially satisfied.
Positive Satisfaction	if $l \leq u \wedge u = n$, tell $c_1, \ldots, c_n$ to the store.
Negative Satisfaction	if $l \leq u \wedge u = 0$, tell $\neg c_1, \ldots, \neg c_n$ to the store.
Positive Reduction	if c_i is true, rewrite the constraint as $\#(l-1, u, \{c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n\})$.
Negative Reduction	if c_i is false, rewrite the constraint as $\#(l, u, \{c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n\})$.
Failure	if $l > u$, the constraint fails.

Table 2.1: Informal rules for evaluation of cardinality constraints

The magic series problem can be expressed by cardinality constraints. For instance, each s_i should satisfy $\#(s_i, s_i, \{s_1 = i, s_2 = i, \ldots, s_n = i\})$. Although Firebird does not support cardinality constraints, it is by no means a limitation, as we shall see shortly.

2.5.1.3 Using Concurrent Processes

We hope to express the magic series problem without resorting to any specialized language constructs. The following constraint, which gives the cardinality of a single equality constraint, is taken from [CCD94].

```
bool(X,Y,B) :- X = Y | B = 1.
bool(X,Y,B) :- X \== Y | B = 0.
bool(X,Y,1) :- X = Y.
bool(X,Y,0) :- X #\= Y.
```

We could find the number of occurrences of an integer N in a list L using the following predicate.

```
occur(X,N,[]) :- X = 0.
occur(X,N,[Y|T]) :- [B] :: 0..1, bool(Y,N,B), X #= X1 + B,
                    occur(X1,N,T).
```

The remaining parts of our magic series program in Firebird are given below.

```
magic(N,L) :- M is N + 1, gen_list(M,L), L in 0..M,
              occurrences(L,0,L).

occurrences([],N,L).
occurrences([X|T],N,L) :- occur(X,N,L), N1 is N + 1,
                          occurrences(T,N1,L).

gen_list(0,L) :- L = [].
gen_list(N,L) :- N \== 0 | L = [H|T], N1 is N - 1, gen_list(N1,T).
```

2.5.2 Weak Queens

The *weak queens problem* is a variation of the n-queens problem. Instead of placing n queens on an $n \times n$ board, a queen is placed on each column of an infinitely long band of width n. A queen can attack horizontally, vertically or diagonally, but the horizontal and diagonal attack range of a queen is only k, $k < n$. Hence a queen in this problem is 'weaker' than a normal queen. Queens can be put on columns $k+1, k+2, \ldots$ For example, if $n = 5$ and $k = 2$, one of the solutions is (2,4,1,3,5,1,...). The following is a Firebird program which solves the weak queens problem.

```
weak_queens(N,K,L) :- K < N | L in 1..N, constraint(K,L).

constraint(K,[]).
constraint(K,[X|T]) :- safe(K,X,T,1), constraint(K,T).

safe(K,X,_,N) :- N > K | true.
safe(K,X,[Y|T],N) :- N =< K |
                     noattack(X,Y,N), N2 is N + 1, safe(K,X,T,N2).

noattack(X,Y,N) :- X #\= Y, X #\= Y + N, X #\= Y - N.
```

We can apply the lazy evaluation technique to this program.

```
consumer(0,L) :- L = [].
consumer(M,L) :- M > 0 | L = [H|T], M2 is M - 1, consumer(M2,T).
```

In the query

```
weak_queens(5,2,L),consumer(6,L).
```

the goal `weak_queens/3` will produce the solution to the first 6 columns one by one upon request from the consumer.

Chapter 3

Operational Semantics

The semantics of Firebird is based on the formalism of [Mah87]. The domain theory D allows constraints of finite set membership over constants.

A domain variable x with domain δ is represented by a variable x and a constraint $x \in \delta$.

A *program* P consists of one or more *clauses*. A clause is of the form

$$H \leftarrow G \mid B, B'$$

where G is a conjunction of ask constraints, B is a conjunction of atoms and B' a conjunction of tell constraints, assuming that all head matchings have been transformed to ask constraints [CG85]. We define a *goal* G to be a multiset A of atoms plus a multiset C of constraints. G and its multiset of atoms and constraints are annotated by a (possibly empty) list σ of non-negative integers and subscripted by an integer i, *i.e.* $G_i^\sigma = (A_i^\sigma \cup C_i^\sigma)$. A *query* is represented as $G_0 = (A_0 \cup C_0)$ annotated by an empty list of non-negative integers.

3.1 The Firebird Computation Model

Definition 3.1 *A* derivation step *using the* Firebird Computation Model *from a goal* $G_i^\sigma = (A_i^\sigma \cup C_i^\sigma)$ *to a goal* $G_j^\tau = (A_j^\tau \cup C_j^\tau)$, *written as*

$$G_i^\sigma \Rightarrow_F G_j^\tau$$

is defined as follows.

1. *If there is a goal* $A \in A_i^\sigma$ *and there is a clause* $H \leftarrow G|B, B'$ *in the program such that* A *can commit to that clause subject to the commit law stated below, then* $j = i+1, \tau = \sigma, A_j^\tau = (A_i^\sigma \setminus \{A\}) \cup B$ *and* $C_j^\tau = C_i^\sigma \cup \{A = H\} \cup G \cup B'$. *This is called an* indeterministic derivation step. *If there is only one clause that* A *can commit to, then* G_i^σ *is called a* deterministic goal *and the derivation step is called a* deterministic derivation step.

2. *Let* $\overline{\delta}$ *denotes the complement of* δ *with respect to the Herbrand Universe. If there is a constraint* $x \in \delta$ *such that* $D \models C_i^\sigma \rightarrow (x \in \delta \wedge x \notin \overline{\delta})$ *and* $\delta = \{a_1, \ldots, a_n\}, n > 1$, *then* $j = 0, \tau = \sigma \cdot k, A_j^\tau = A_i^\sigma$ *and* $C_j^\tau = C_i^\sigma \cup (x = a_k), 1 \leq k \leq n$ *provided that* $C_i^\sigma \wedge (x = a_k)$ *is consistent. Here* '$\cdot$' *is the 'append' operator. This is called a* nondeterministic derivation step.

3. *The indeterministic derivation step and the nondeterministic derivation step are the only derivation steps allowed. If both the indeterministic derivation step and the nondeterministic derivation step are applicable, only one of them will be selected.*

3.2 The Firebird Commit Law

Following the flat GHC convention, the commit law of Firebird is simpler than that of ALPS [Mah87]: a clause $H \leftarrow G \mid B, B'$ can be committed to only if the clause is *validated.* In the other words, given an atom A selected from A_i^σ, we must find a clause with a head and ask constraints which satisfies the following condition.

$$D \models \forall x_g (C_i^\sigma \rightarrow \exists x_l (A = H \wedge G))$$

Here, x_g and x_l denote the variables in G_i^σ (the 'global' variables) and the variables local to the clause respectively. Likewise, a clause is *invalidated* if

$$D \models \neg\exists (A = H \wedge C_i^\sigma \wedge G)$$

3.3 Derivation

Definition 3.2 *A* derivation *from a goal G_i^σ to a goal G_j^τ, written*

$$G_i^\sigma \stackrel{*}{\Rightarrow}_F G_j^\tau$$

is a (possibly empty) sequence of derivation steps such that either

$$\sigma = \tau$$

$$i = j$$

or there exists goals $G_{i_1}^{\sigma_1}, G_{i_2}^{\sigma_2}, \ldots, G_{i_n}^{\sigma_n}$ such that

$$G_i^\sigma \Rightarrow_F G_{i_1}^{\sigma_1} \Rightarrow_F G_{i_2}^{\sigma_2} \Rightarrow_F \ldots \Rightarrow_F G_{i_n}^{\sigma_n}$$

and $i_n = j$ and $\sigma_n = \tau$. G_j^τ is said to be derivable *from G_i^σ.*

3. $A'' = \emptyset$.

$$P^*, D \models G_0 \leftrightarrow \bigvee_{i=1}^{\infty} \exists y_i(C_i)$$

where $C_1, C_2, \ldots$ *are final constraint sets associated with the successful derivations, and* y_i *are the variables local to* C_i[1].

Proof We shall prove this lemma using mathematical induction on the height h of a Firebird search tree.

Base case: $h = 1$. The Firebird search tree is in either of the following two forms.

1. If a deterministic derivation step is applied on $G_0 = (A_0 \cup C_0)$ using a completed clause $H \leftrightarrow G \mid B'$, then it has only one child $G_1 = (\emptyset \cup C_1)$ and $C_1 = (C_0 \wedge G \wedge A_0 = H \wedge B')$. Since $D \models \forall x_g(C_0 \rightarrow \exists x_l(A_0 = H \wedge G))$, we have $P^*, D \models G_0 \leftrightarrow \exists y_1(C_1)$ where y_1 are the variables local to C_1.

2. If a nondeterministic derivation step is applied on $G_0 = (\emptyset \cup C_0)$, it will have n children $G_0^1 = (\emptyset \cup C_0^1), \ldots, G_0^n = (\emptyset \cup C_0^n)$ such that for some variable x, $D \models C_0 \rightarrow (x \in \{a_1, \ldots, a_n\} \wedge x \notin \overline{\{a_1, \ldots, a_n\}})$ and $C_0^i = ((x = a_i) \wedge C_0), 1 \leq i \leq n$. Clearly we have $P^*, D \models G_0 \leftrightarrow (\exists y_1(C_0^1) \vee \cdots \vee \exists y_n(C_0^n))$, where the y_i's are the variables local to C_0^i.

Induction hypothesis: the lemma is true for $1 \leq h \leq k$.

Induction step: $h = k + 1$. Consider a Firebird search tree of height $k + 1$.

1. If a deterministic derivation step is applied on $G_0 = (A_0 \cup C_0)$ using a completed clause $H \leftrightarrow G \mid B, B'$, then it has only one child $G_1 = (A_1 \cup C_1)$

[1] P^* is the Clark's completion [Cla78] of program P.

where $A_1 = ((A_0 \setminus \{A\}) \wedge B)$ and $C_1 = (C_0 \wedge G \wedge (A = H) \wedge B')$. Since $D \models \forall x_g(C_0 \rightarrow \exists x_l((A = H) \wedge G))$, we have $P^*, D \models G_0 \leftrightarrow G_1$. Now that the subtree rooted at G_1 is of height less than or equal to k, we have

$$P^*, D \models G_1 \leftrightarrow \bigvee_{i=1}^{\infty} \exists y_i(C_i)$$

Hence

$$P^*, D \models G_0 \leftrightarrow \bigvee_{i=1}^{\infty} \exists y_i(C_i)$$

where $C_1, C_2, \ldots$ are final constraint sets associated with the successful derivations, and y_i is the variables local to C_i.

2. If a nondeterministic derivation step is applied on $G_0 = (A_0 \cup C_0)$, then it has n children $G_0^1 = (A_0^1 \cup C_0^1), \ldots, G_0^n = (A_0^n \cup C_0^n)$ such that for some variable x, $D \models C_0 \rightarrow (x \in \{a_1, \ldots, a_n\} \wedge x \notin \overline{\{a_1, \ldots, a_n\}})$ and $C_0^i = (x = a_i \wedge C_0), A_0^i = A_0, 1 \leq i \leq n$. Therefore we have $P^*, D \models G_0 \leftrightarrow (G_0^1 \wedge \cdots \wedge G_0^n)$. Now as subtrees rooted at $G_0^1, \cdots, G_0^n$ are of height less than or equal to k, we have

$$P^*, D \models G_0^i \leftrightarrow \bigvee_{j_i=1}^{\infty} \exists y_{j_i}(C_{j_i})$$

where C_{j_i}'s are final constraint sets associated with the successful derivations in the subtree rooted at G_0^i, and y_{j_i}'s are the variables local to C_{j_i}'s. $1 \leq i \leq n$. By the distributive properties of disjunction, we have

$$P^*, D \models G_0 \leftrightarrow \bigvee_{i=1}^{\infty} \exists y_i(C_i)$$

where $C_1, C_2, \ldots$ are final constraint sets associated with the successful derivations, and y_i is the variables local to C_i.

Theorem 3.1 (Soundness of Firebird Computation Model) *Given a program P, if a query $G_0 = (A_0 \cup C_0)$ has a successful derivation with final constraint set C, then*

$$P, D \models C \rightarrow A_0 \cup C_0$$

Proof This is a direct consequence of Lemma 3.2 and the definition of successful derivation.

Theorem 3.2 (Conditional Completeness of Firebird Computation Model) *Given a program P and a query $G_0 = (A_0 \cup C_0)$, if $P^*, D \models C \rightarrow A_0 \wedge C_0$, and for every goal $G'' = (A'' \cup C'')$ derivable from G_0, one of the following condition holds,*

1. *G'' is a deterministic goal.*
2. *a nondeterministic derivation step is applied to G''.*
3. *$A'' = \emptyset$.*

then G_0 has a successful derivation with a final constraint set C_i' such that $P^*, D \models C \rightarrow \exists y_i(C_i')$.

Proof By Lemma 3.3 we have

$$P^*, D \models G_0 \leftrightarrow \bigvee_{i=1}^{\infty} \exists y_i(C_i)$$

where $C_1, C_2, \ldots$ are final constraint sets associated with the successful derivations, and y_i is the variables local to C_i, under the conditions as specified in the lemma. Hence we have

$$P^*, D \models C \leftrightarrow \bigvee_{i=1}^{\infty} \exists y_i(C_i)$$

4.1 An Illustrative Example

To illustrate how Firebird exploits data-parallelism, it is helpful to trace the execution of 5-queens using the program presented before and the query `queen(5,[X1,X2,X3,X4,X5])`. At the first deadlock, the system has the following suspended constraints. All the other goals have been reduced, and all variables have the same domain {1,2,3,4,5}.

```
X1 ≠ X2   X1 ≠ X2 + 1   X1 ≠ X2 - 1
X1 ≠ X3   X1 ≠ X3 + 2   X1 ≠ X3 - 2
X1 ≠ X4   X1 ≠ X4 + 3   X1 ≠ X4 - 3
X1 ≠ X5   X1 ≠ X5 + 4   X1 ≠ X5 - 4
X2 ≠ X3   X2 ≠ X3 + 1   X2 ≠ X3 - 1
X2 ≠ X4   X2 ≠ X4 + 2   X2 ≠ X4 - 2
X2 ≠ X5   X2 ≠ X5 + 3   X2 ≠ X5 - 3
X3 ≠ X4   X3 ≠ X4 + 1   X3 ≠ X4 - 1
X3 ≠ X5   X3 ≠ X5 + 2   X3 ≠ X5 - 2
X4 ≠ X5   X4 ≠ X5 + 1   X4 ≠ X5 - 1
```

Figure 4.1: Constraints remaining after first deadlock

If we label `X1` and try the 5 possible values in a data-parallel fashion, we can evaluate the first 12 constraints with an ideal 5 times speedup on a SIMD machine. A second deadlock will occur. If each branch chooses to create a choice point on `X2`, there will be 3+2+2+2+3 = 12 branches (see Figure 4.2). Thus the next 9 constraints can be solved with 12 times speedup. `X3` will also be labeled, giving rise to a total of 14 branches. Therefore, the remaining 9 constraints can be evaluated with 14 times speedup. Thousands of processor elements can be fully utilized easily in this way because many problems are combinatorial in nature.

5.1 Basic DPAM

5.1.1 Hardware Requirements

Different data-parallel computers have different capabilities. Therefore, the minimum hardware requirements are defined in such a way that DPAM can be implemented on most of the recent data-parallel computers. The hardware consists of the following components:

1. a *processor element array*, consisting of a number of identical *processor elements*, executing in a synchro-parallel manner, each with its local memory, known collectively as *vector memory*. In a memory-read operation, each processor element may access a different memory location (known as *local indirect addressing*), but in a memory-write operation, all processor elements must access the same location[1]. There is no inter-processor communication. Each processor element has a *mask* bit. A processor element executes instructions if and only if its mask bit is set, except that there are special instructions which move a bit to the mask bit uncontingently.

2. a *host computer*, which is responsible for dispatching instructions to all processor elements. The host computer has its own store called *scalar memory*. The host computer may broadcast a data item to all processor elements whose mask bits are set or receive a data item from an arbitary processor element. The host computer must also have the ability to check

[1]Some data-parallel computers have faster memory access if all processor elements access the same memory location (e.g. Maspar MP-1, Connection Machine CM-2, some vector supercomputers like Hitachi S-820). We impose this restriction because it leads to performance improvement on such computers. Furthermore, it is expected that more real-world machines can be mapped to a more restrictive abstract machine.

if none of the processor elements have their mask bits set, which is useful for conditional branching.

The above features are directly available or can be emulated on most of the recent SIMD computers. Since different machines has different interprocessor communication topologies, DPAM does not require any form of interprocessor communication except moving data to and from the host computer. Likewise, since some machines (*e.g.* supercomputers) have a single, shared memory space for all processor elements whereas some have memory distributed over the processor elements, we assume that the memory is distributed. Nevertheless, some older machines, including Cray-1, do not have local indirect addressing and hence do not meet our minimum hardware requirement.

5.1.2 Procedure Calling Convention And Process Creation

In DPAM, a *procedure* is defined to be an executable subroutine. An n-ary goal or constraint is compiled to a procedure which takes n input arguments, which are stored in general purpose vector registers p1 to pn. The *mask* (Section 4.3) bit vector defines the set of physical partitions in which the procedure is applicable.

In WAM [AK91], failure is like an exception. When a failure occurs execution continues directly at the next clause. This is not possible in a data-parallel system because the failure of a physical partition does not imply the failure of all physical partitions. Therefore, upon return, the procedure signifies that a physical partition has succeeded by setting the mask bit and that a physical partition has failed by clearing the mask bit.

In Figure 5.1, we illustrate how the mask bit vector is changed during execution of 3 call instructions on a 8-processor machine. Note that the mask shrinks monotonically.

procedure call	input mask	output mask
`call p/0`	11111111	11011111
`call q/0`	11011111	01011001
`call r/0`	01011001	00011000

Figure 5.1: Example: mask bit vector

The *concurrent process scheduler*, or simply *scheduler*, is responsible for executing procedures. The scheduler treats all procedures as black boxes. It simply performs a subroutine call to the procedure after loading the input arguments. A procedure may call other procedures in the same manner.

If a procedure (a compiled goal or constraint) wants to suspend, it calls a system library subroutine `suspend_process` which saves all the argument registers and other status information (*e.g.* continuation pointer) in a *process structure*. The procedure must tell `suspend_process` the set of physical partitions which need suspension by setting the mask bit vector. Then the procedure returns control to the caller with the mask bit set to the set of physical partitions which have either succeeded or suspended. In other words, the caller cannot distinguish whether the callee has succeeded or suspended (the scheduler can, of course).

The evaluation of tell constraints is left to the particular implementation. DPAM defines only the interface—the constraint must obey the abovementioned calling convention, and it must call `suspend_process` for a suspension.

In DPAM, ask constraints never appear in any process queues. They do not

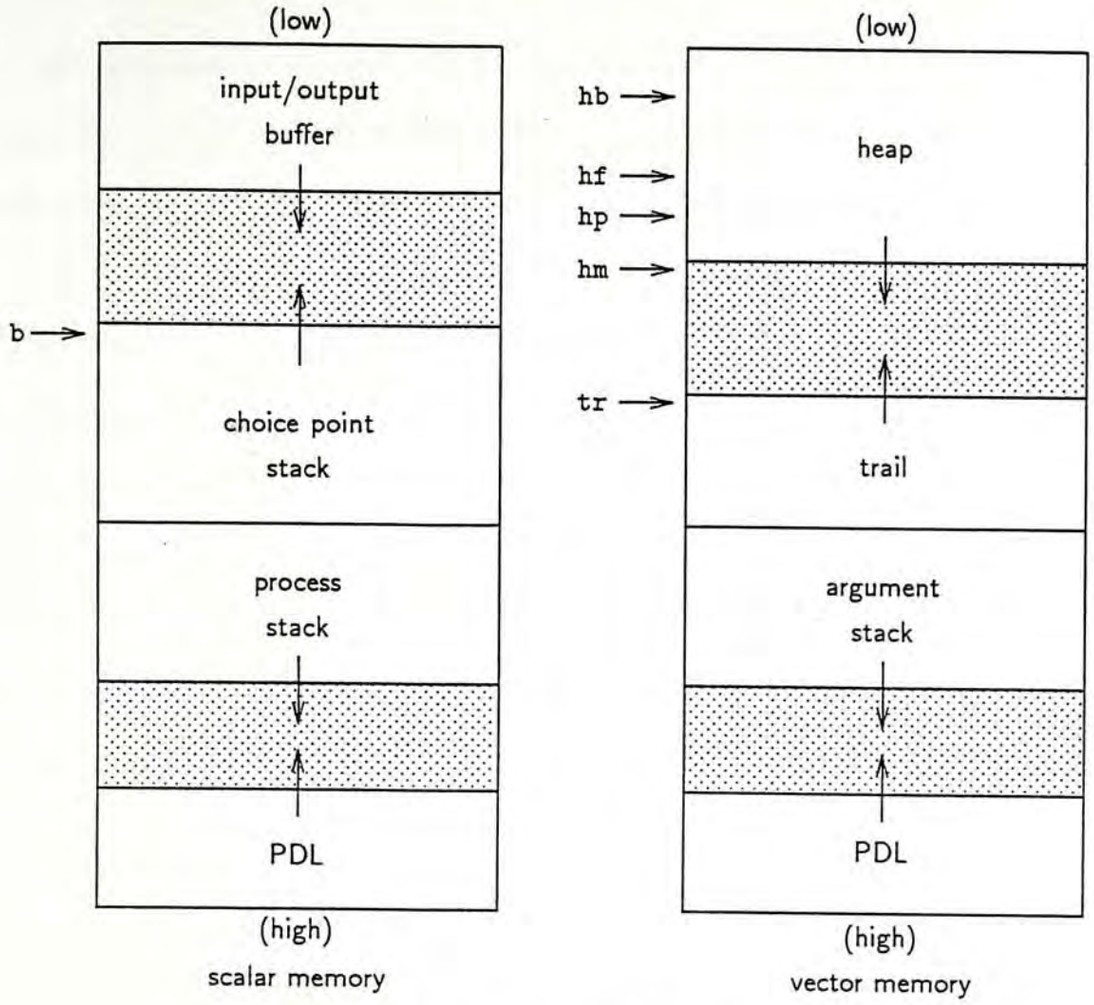


Figure 5.2: Memory areas and scalar registers

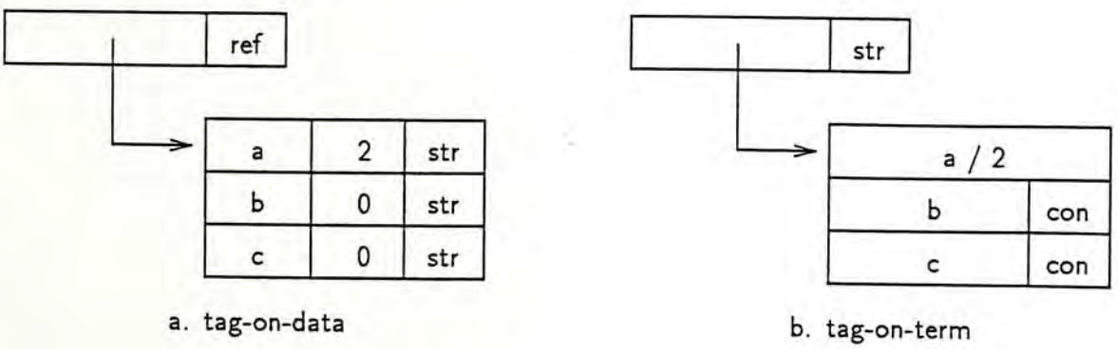


Figure 5.3: Example: tag-on-term and tag-on-data representations of `a(b,c)`

◇ pointer to bit vector representation of the domain.

The bit vector is accessed via a pointer so that any changes are backtrackable (this will be explained later in Section 5.2.2.2). The pointer to the bit vector always points to the address which would have contained the value 0 even if 0 is outside the range of the domain. For the example in Figure 5.5, the domain of the variable is 0..159 and the bit vector is stored at address 120 (hex). The pointer to the bit vector field will be 120 (hex) because the bit representing the value 0 is stored at 120 (hex). In this way, there is no need to update the bit vector and its pointer when the minimum and maximum fields are changed. For instance, if the minimum is changed to 96, the pointer to bit vector field remains to be 120 (hex), although the locations 120–12b (hex) have become garbage.

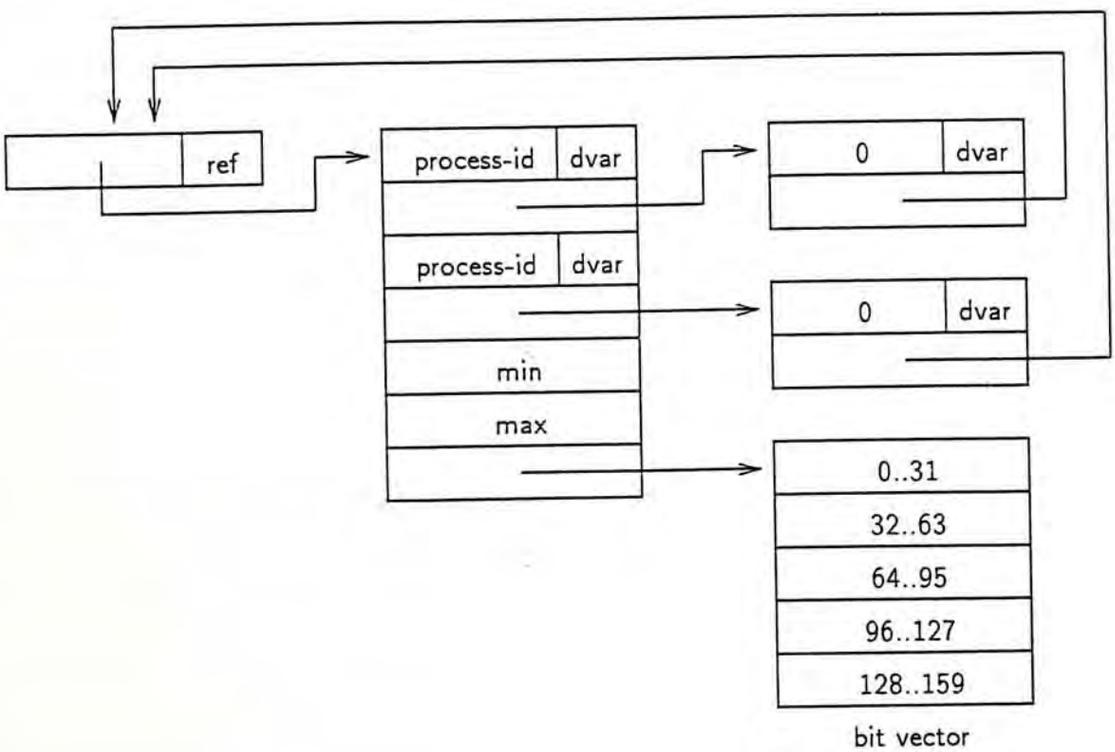


Figure 5.5: Example: a *d*-variable

Unlike the memory write operation, the ability for each processor element to read a different memory location in parallel is mandatory. Consider the dereferencing of a vector of reference chains, for instance.

5.1.4 Registers

The host computer has the following scalar registers in addition to a file of general purpose scalar registers `c1` to `cn` (see Figure 5.2 again).

`hp` heap pointer

`hf` heap frame pointer

`hm` heap pointer maximum

`hb` heap pointer at last choice point

`tr` trail pointer

`b` current choice point

`ip` instruction pointer

`cp` continuation pointer (keeps return address of procedure invocation)

Vector registers include the *process count* (`c`) and the *status word* (`sw`), in addition to a number of general purpose vector registers (`p0` to `pm`).

5.1.5 Process Management

5.1.5.1 Process Structure

A *process structure* must contain enough information for the reinvocation of a process when the process is resumed. In DPAM, the process structure is scalar and resides on the host computer. The arguments are stored separately in the vector argument stack. The process structure consists of the following fields.

- ◇ process-id
- ◇ continuation pointer
- ◇ number of arguments
- ◇ pointer to the argument vectors on the argument stack
- ◇ pointers to the next and the previous process structures
- ◇ pointers to the next and the previous processes in a suspension queue (to be discussed in Section 5.2.2.1)

To save storage, the mask of a process is indicated by its first argument vector. A physical partition is outside the mask of a process if it has a zero first argument. A pseudo-argument vector which stores the mask will be added to 0-ary processes.

5.1.5.2 Overview of System Queues

The system maintains *hash queues*, the *ready queue*, the *resumption queue* and *free lists* for the creation and scheduling of processes. Both the next and previous pointer fields are used in *hash queues*, which are doubly linked circular lists used to locate a process with a given process-id in a resumption operation. Alternatively, the next and previous pointers can be used as *primary* and *secondary linkage pointers* so that a process can be in two singly linked lists simultaneously. The *ready queue*, the *resumption queue* and *free lists* use the primary linkage pointer and the *labeling queue* uses the secondary one.

Each process in the ready queue is executed and then the process structure is placed back in one of the free lists. When a process is created, DPAM tries to reuse process structures in the free lists first before it allocates new space

on the process stack and the argument stack. When a process is resumed it is first placed in the resumption queue. After the ready queue is exhausted the processes in the resumption queue will be moved to the ready queue.

5.1.5.3 Suspension and Resumption

There are two possible ways of implementing suspension and resumption.

Method 1 If there is a suspension, a new process structure will be created, and a *process-id* will be assigned to it. The new process will be placed in one of the hash queues. Its process-id will be stored in a suspension list node which becomes the head of the suspension list of the unbound variable. New suspension list nodes can be inserted at the head of the suspension list, by copying the old list head to the top of the heap and replacing the old list head by the newly created node (see Figure 5.7).

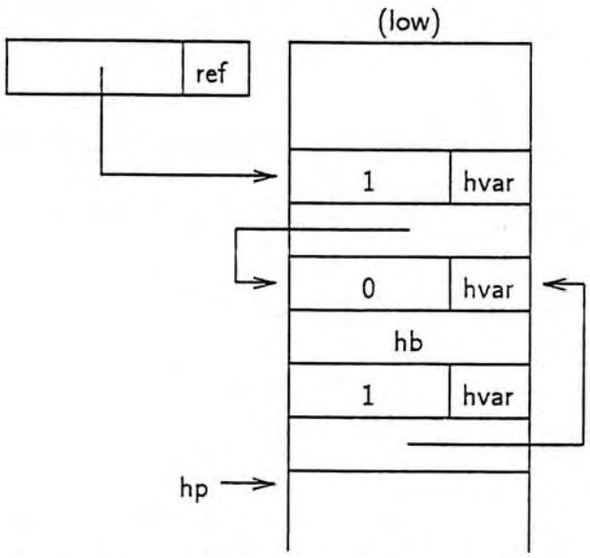
When a variable is assigned, each process-id in the suspension lists is fetched. If a process can be found in one of the hash queues, it is removed from the hash queue and added to the resumption queue. Otherwise, the process has already been resumed by another assignment and can be ignored.

Naturally, each processor element may have a suspension list with different processes. To resume them the following algorithm is devised.

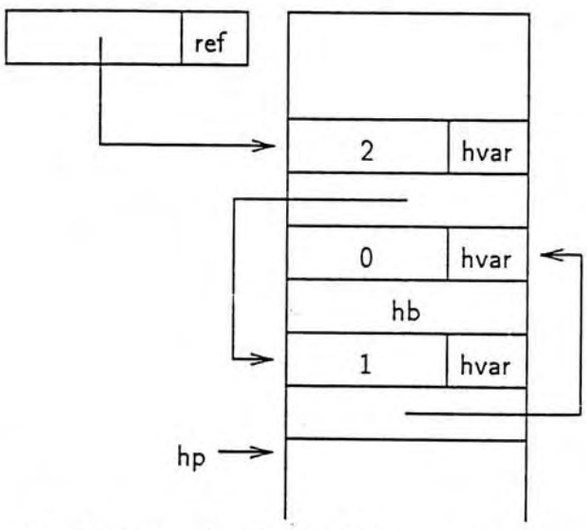
1. Select an arbitrary processor element G.

2. Each processor element fetch the process-id in the head of the suspension list, unless the list is empty.

3. Let the process-id as obtained from step 2 by processor element G be i. Resume the process with process-id i.



1. suspension list head copied to top of heap



2. old suspension list head is overwritten

Figure 5.7: Suspension

4. Every processor element whose process-id in the head of suspension list is equal to i may proceed to the next suspension list node.

5. Repeat steps 2–4 until the entire suspension list of parition G has been traversed.

6. Select another processor element G and repeat steps 2–5 until the suspension list of every processor element has been traversed.

See Figure 5.8 for an example of the above algorithm at work.

Method 2 The traditional scheme using no hash queues can be used. Instead of a process-id, the head of each suspension node contains the pointer to a scalar hanger [Cra90a] residing on the host. The hanger will point to the process structure. When a process is resumed the hanger will be zeroed to prevent the process from being resumed more than once. An algorithm similar to that of method 1 can be used to resume a vector of suspension lists.

The first method will be slower because of searching but it makes simpler memory management possible. This is because it is not necessary to write a separate garbage collector to clear obsolete hangers from the host's memory. Both schemes allow old process structures to be freed and reused.

To resume a process, we can do either of the following.

1. The process is resumed in all physical partitions even if the variable is assigned in some physical partitions only.

2. The process is split into two. One of the processes, with the mask set to consist of the physical partitions in which the resumption has taken place, will be placed in the resumption queue. The other process, consisting of

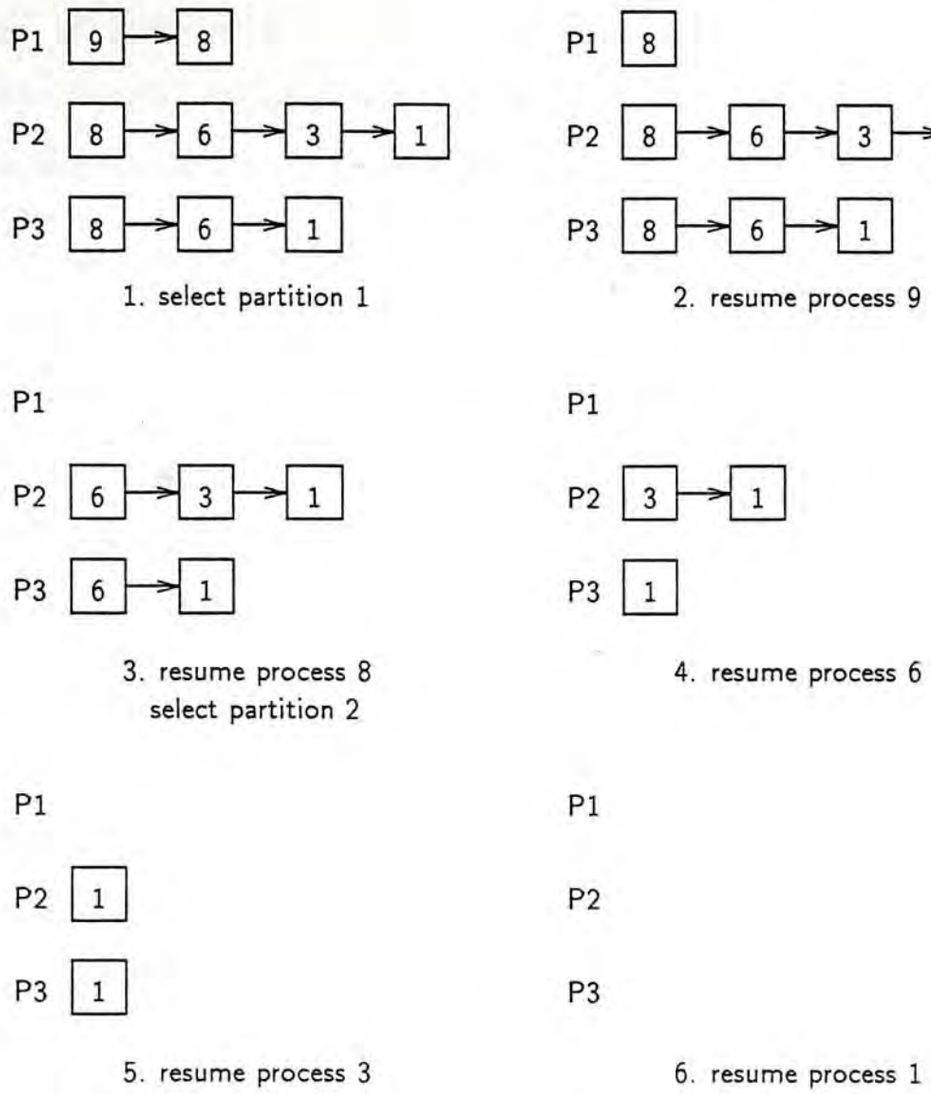


Figure 5.8: Example: resumption of suspension lists

physical partitions in which the resumption has not taken place, will be left in the suspension list.

We took the first method because it is simpler and does not have the overhead of splitting. On a SIMD computer, the time to execute a process is unrelated to the number of active processor elements. The falsely resumed physical partitions will execute the process alongside with the others without any additional cost. If these physical partitions commit we actually save execution time. Otherwise,

they will just re-suspend.

Unlike *Parallel Parlog* [Cra90a], we do not make any distinction between *single-suspension* and *multi-suspension* because

1. The Parallel Parlog scheme improves the efficiency of *single-suspension*, but for finite domain constraints, multi-suspension is more the rule than the exception. All unary constraints in the form X *op* n, where *op* is a equality, inequality or disequality relation, can be solved immediately without suspension. Other constraints suspend on two or more variables (*multi-suspend*).

2. In Parallel Parlog, the pointer to the next process structure resides in the process structure itself in a single-suspension. However, we have a vector of variables in a variable assignment, resulting in several threads of processes to be resumed in general. Thus the *next* field of the DPAM process structure, which is scalar, cannot be used.

5.1.5.4 Deadlock Detection

A *labeling process* is responsible for labeling a *d*-variable in a nondeterministic derivation step. It is created and placed in the *labeling queue* whenever a process suspends on a *d*-variable for the first time, unless

1. the *d*-variable is created by `::/2`, or

2. the *d*-variable is originally a *h*-variable whose domain is inferred from an equality constraint. For example, suppose X is a *h*-variable and there is a constraint $X = Y + 1, Y \in \{1, 2\}$. It is sufficient to label Y only but a labeling process for X is never created.

In order to detect deadlock of individual physical partitions, a process count and a deadlock flag are maintained in each physical partition. The process count of a physical partition keeps track of the number of processes in that physical partition. The deadlock flag is a bit in the status word which is initially set. When a process is resumed in some physical partitions, the deadlock flags of those physical partitions are cleared. When the ready queue is exhausted, the physical partitions are checked for deadlock.

1. If the process count of any physical partitions are zero, those physical partitions have succeeded.

2. If there are any physical partitions which have not succeeded, and any of them have set deadlock flags, a deadlock has occured. A process is moved from the labeling queue to the ready queue. If the labeling queue is empty, the deadlock is irrecoverable.

3. If the resumption queue is empty, a deadlock has occured. A process is moved from the labeling queue to the ready queue. If the labeling queue is empty, the deadlock is irrecoverable.

4. After checking, append all processes in the resumption queue to the ready queue. (In other words, if a deadlock has occured, the ready queue will consist of a labeling process followed by other resumed processes. Otherwise, the ready queue will consist of resumed processes only.) Set deadlock flag for all physical partitions again.

5.1.6 Unification

Following JAM [Cra90b] (an abstract machine for the parallel execution of Parlog [CG86]) and WAM [AK91], when two unbound *h*-variables, X and Y are unified, the one which is created later is bound to the one which is created first. In WAM, this age comparison reduces the chance of trailing and avoids the binding of a heap variable to a stack variable[4]. However, in JAM and DPAM, the age comparision ensures the correct operation of the $X = Y$ ask constraint (where both X and Y are unbound). $X = Y$ will suspend only on the variable created later.

If a *h*-variable is unified with a *d*-variable, the *h*-variable is bound to the *d*-variable. Therefore, if a process asks whether $X = Y$ where X is a *h*-variable and Y is a *d*-variable, the process should suspend on X.

5.1.7 Variable Table

In JAM [Cra90b], when the truth value of a guard cannot be determined because of an unbound variable, that variable is stored temporarily in a *variable table*. If a commitment to another clause is made. the variable table can simply be ignored. If a suspension is really needed for the process, it will suspend on all variables in the variable table. The variable table is a fixed array.

The same concept is adopted in DPAM. However, each entry in the variable table is extended to a <*mask, variable*> pair. The DPAM instruction suspend

[4]A *stack variable*, as mentioned in another footnote before, is a self-referential pointer in the environment stack, saving the space of 1 heap cell. A heap variable is a self-referential pointer on the heap. Since a heap variable is permanent but a stack variable is freed after the procedure completes its execution, a heap variable must not be allowed to be bound to a stack variable.

stores a variable in the variable table together with the current mask. The variable table is *not* a fixed array, but resides on the vector PDL[5]. The following example illustrates the necessity of this.

```
a(1,Y) :- b(Y).
a(X,1) :- b(X).

b(x).
b(y).
```

When a/2 is executed, some physical partitions will commit to the first clause (let P_1 be the set of such physical partitions). The others, however, may potentially suspend on Y and therefore Y is stored in the variable table (let the set of these physical partitions be P_2). Next, b(Y) will be executed, with the mask set to P_1. Note that b/1 maintains its own variable table which must not have any conflicts with that of a/2. Hence the vector PDL is chosen as the place to store the variable table.

5.2 DPAM with Backtracking

Like any or-parallel logic programming system, the number of parallel branches explodes combinatorially in Firebird. For example, more than 2,000 processor elements are required to solve 8-queens. When the processor elements are exhausted, the system resorts to *parallel backtracking*, in which the partitions create choice points and backtrack in parallel.

In general, the number of or-branches is different for each partition because the domain size of the domain variable used to set up the choice point varies

[5]In WAM terminology [AK91], the PDL is a stack for miscellaneous purposes. Please see Figure 5.2 on page 35 again.

from partition to partition. As a result, some partitions will finish attempting all the alternatives earlier than the others.

We adopt the *synchronous backtracking* approach, in which the partitions which have finished wait until all partitions have finished before going back to the last choice point. The advantage of this approach is lower trail and choice-point management overhead. If the finished partitions go back to the last choice point immediately, a higher degree of parallelism may result but we have not yet implemented it to evaluate it empirically.

All partitions will resort to backtracking if any of the partitions do not have enough processor elements. In this way, more parallelism can be exploited if there is a nondeterministics derivation step afterwards with domain sizes small enough so that backtracking is unnecessary. In synchronous backtracking, a physical partition will be idle after all its alternatives have been attempted. As a result, even if we exploit or-parallelism in partitions with enough processor elements, such partitions will still have to wait for those partitions using backtracking.

Like WAM [AK91], a choice point is freed before the last possible value is attempted (since only one choice is left, the system may 'commit' to that choice). Whereas WAM has `try_me_else`, `retry_me_else` and `trust_me` instructions to create, update and remove a choice point, such instructions does not exist in DPAM. A choice point is created by the labeling process if needed and updated/removed by the scheduler.

We shall present trailing and the recovery of process queues in detail. Both of them has only one objective—to restore the machine to its original state when backtracking occurs. The reader is encouraged to verify herself/himself whether

this objective is met in our designs.

5.2.1 Choice Point

A choice point is created when there are not enough processor elements to be divided among the alternatives in a nondeterministic derivation step. The scalar register `b` points to the top of the (scalar) choice point stack. A choice point contains the following fields:

- ◇ process-id allocation counter at last choice point
- ◇ ready queue at last choice point
- ◇ labeling queue at last choice point
- ◇ trail pointer (`tr`) at last choice point
- ◇ `hb` at last choice point
- ◇ restore queue
- ◇ head and tail of suspension queue

The process count, status word and processor element allocation information are saved on top of the heap rather than the choice point stack because they are vectors. They can be accessed at any time via `hb`. The consumed heap space can be reclaimed when the choice point has finished.

5.2.2 Trailing

The trailing scheme of WAM [AK91] is extended for use in a data-parallel context. The trail stack resides in the vector memory of the processor element array. Each entry in the trail is composed of a <*mask, address, old value*> triplet. The

scalar `tr` points to the top of the trail stack. The `hb` (heap pointer at last choice point), which is scalar like `hp`, is used to determine whether a variable should be trailed. If trailing is not needed by any physical partition then no trail entry is created on the trail stack. Otherwise the mask field, which is set to the set of physical partitions which require trailing, is recorded together with the address and old value fields on the trail.

5.2.2.1 Trailing of Suspension List Updates

Several processes may suspend on an unbound variable in between two choice points. As a result, the unbound variable (or the suspension list and touch list fields of a domain variable) may be updated several times, which may lead to *multiple trailing*. Multiple trailing refers to the problem that a memory location is trailed more than once in between two consecutive choice points. This is inefficient because only the first trail entry is sufficient for restoring the heap to its original state when backtracking occurs.

We observe that when a new node is inserted at the head of a suspension list, the first suspension list node is copied to the top of the heap. Then it is overwritten by the new suspension list node (see Figure 5.7 on page 44 again). The link field of the new node provides a clue to the age of the variable. At the next time the variable is updated, trailing is required only if it points to an address below `hb`. In other words, when a variable is bound or a new suspension list node is added, it is the link field of the first suspension list node rather than the address of the suspension list node itself which is compared with `hb`. The empty node of a freshly created unbound variable has a zero *process-id* as head and `hp` at the time of creation as tail. Table 5.1 compares the operations

required by a system using time stamps and one which does not.

operation	with time stamp	without time stamp
variable creation	read current time stamp create time stamp field create empty suspension list	 create empty suspension list
update list	read suspension list resume read time stamp check time stamp trail time stamp update time stamp copy node to top of heap trail node update node	read suspension list resume check address copy node to top of heap trail node update node
dereferencing	unaffected	unaffected
unwind	copy time stamp and node	copy node only

Table 5.1: Comparison of suspension list update with and without time stamp

It is evident that efficiency can be improved no matter trailing is needed or not. In addition, we save the memory used for storing time stamps.

This age comparision is different from the age comparison in the unification of two unbound h-variables (see Section 5.1.6, page 49 again). They are compared as follows.

- To determine whether to suspend on X or Y in an ask constraint $X = Y$ (or to determine whether X or Y should be bound to the other in a tell constraint $X = Y$), the time of creation of the variables is considered. Therefore, it is the reference pointers to the unbound cells which are compared.
- To avoid multiple trailing, the time of last modification is considered. Therefore, the link field of the first suspension list node is compared with

hb.

5.2.2.2 Trailing of Domain Variables

We do not trail the whole bit vector as in [DC93], but create a new bit vector on the top of the heap, modify the bit vector pointer field of the domain variable and trail the modification of the pointer instead. Like the trailing of a suspension list update, the bit vector pointer field itself is compared with hb to check if trailing is necessary.

In the time-stamp approach, the time-stamp itself must be trailed as well, and the bit vector is copied twice (when it is trailed and when the trail is unwound). In our approach, however, the bit vector is copied only once. We saved trail space, but at the expense of heap space. In both cases the memory can be reclaimed upon backtracking.

There are no pointers in the minimum and maximum fields of a *d*-variable which can be used to infer the age. Therefore, unlike suspension list or bit vector updates, time stamps are associated with the minimum and maximum fields in order to avoid multiple trailing. We use the value of hp at the last time the field is updated as the time stamp.

When the domain of a domain variable is modified, the touch list (discussed in Section 5.1.3.2) will be resumed and replaced by an empty list. The process-id field is set to zero as usual, and the link field is set to hp at the time of modification. As a result, the information on when the touch list field of a domain variable is last modified is captured, regardless of whether the modification is an insertion at the head or a deletion of the whole list.

5.2.2.3 Correctness Issues

Consider the following scenario.

1. An invalid value is removed from the domain of a domain variable. As a result, the processes in the touch list of the variable are resumed. After the resumptions the touch list field of the variable is cleared, and replaced by an empty suspension list node with a zero process-id and the current `hp` as the link field, as an indication of the time of the lastest update.

2. No heap space is consumed afterwards, and a choice point is created. Note that `hp` = `hb`.

3. A process suspends on the touch list of the domain variable. Since the link field of the first node is equal to `hb`, the update is not trailed, leading to incorrect result.

Therefore, at least one of the following conditions must be met to ensure correct operation of our trailing mechanism.

1. The update is ultimate (*e.g.* binding a domain variable to a constant), or

2. The update consumes heap space (*e.g.* inserting a node at the head of a suspension list (see Figure 5.7, page 44 again) or creating a new bit vector and updating the pointer to bit vector field of a domain variable).

DPAM enforces condition 2 and ensures correct operation by storing partition allocation information, the status word and the process count on the heap (see Section 5.2.1 on page 52 again) when a choice point is created.

5.2.3 Recovering the Process Queues

The system maintains a *process-id allocation counter.* When a process is created its process-id is set to the process-id allocation counter, and then the counter is incremented. The process-id provides processes with a convenient age. Processes can be trailed like heap terms: at each choice point the process-id allocation counter is saved. If a ready process has an *id* greater than *process-id allocation counter at last choice point*, then it is created after the choice point and can be discarded and returned to the free lists after being executed. Otherwise it must be retained.

Apart from the linkage pointers no fields in the process structures are modified in DPAM. Therefore to recover the ready queue and the labeling queue upon backtracking we just record the heads of the queues in the choice point. If a process suspended before the last choice point is resumed and removed from a hash queue, it should be restored to the hash queue upon backtracking. The *restore queue* is a singly linked list in each choice point consisting of these processes. When a process which suspended before the last choice point is resumed, it is added to both the resumption queue as usual and the restore queue as well. The restore queue uses the secondary linkage pointer. Hashing on the process-id, processes in the restore queue can be returned to the appropiate hash queue.

If a process suspends after the last choice point and has not resumed, the process should be removed from the hash queue and freed upon backtracking. In each choice point there is a *suspension queue* which is a doubly linked circular list of all processes suspended after the last choice point. When a process suspended after the last choice point is resumed it is removed from the suspension queue.

Chapter 6

Implementation

6.1 The DECmpp Massively Parallel Computer

The Firebird language is implemented on a DECmpp [Bla90], which consists of a front-end UNIX workstation and a back-end *data-parallel unit.* The data-parallel unit in turn consists of an *array control unit* (ACU), a *processor element array* (PE) and an inter-processor communication network which supports both mesh and arbitrary communication patterns. The ACU dispatches a single instruction stream to the processor elements. In addition, it broadcasts data to the processor elements and receives the logical or-ing of data from them. A processor element may choose to execute or ignore an instruction based on its *contingent bit.* Each processor element has its own local memory. Memory operations can be *overlapped* with computation. To achieve this the processor elements maintain FIFO queues to store pending memory requests. The system will stall if any instruction cannot proceed because it depends on the result of a memory instruction which has not completed.

A scalar[1] pointer may point to a scalar memory location or a vector memory location. In the latter case, each processor element reads from the same address when the pointer is dereferenced. A vector pointer always points to vector memory locations. Each processor element reads from (or writes to) a different address. This is termed *local indirect addressing.* It is 2–3 times slower than a normal memory operation[2].

6.2 Implementation Overview

In our implementation, both the ACU and PE array are responsible for program execution. ACU serves as the host computer of the data-parallel abstract machine. The front-end workstation is used only for input and output. Since each processor element is very small and slow[3], parallelism is easily counterbalanced by the slow execution of the processor elements. Memory operations are particularly slow. When they cannot be avoided, we try to make use of the architectural feature of DECmpp and overlap them with non-memory instructions. In addition, we abandon interpretation and write a native code generator to convert DPAM to assembly code. Despite all such efforts, the machine is still too slow for our implementation to outperform a good emulator implementation of Prolog on programs where no or-parallelism can be exploited, such as reverse

[1]In the terminology of DECmpp, registers and memory in the ACU are called *singular* and those in the PE array, *plural.*

[2]A 32-bit indirect memory load operation on the processor element array without overlapping takes over 200 clock cycles on DECmpp 12000 Sx-100. The machine runs at 12.5Mhz. DECmpp 12000 Sx-200 (equivalent to MasPar MP-2) is about twice as fast on the same operation.

[3]For example, a 32-bit register to register addition requires about 24 clock-cycles, and a 32-bit multiplication requires more than 200 clock cycles on DECmpp 12000 Sx-100 (The machine runs at 12.5 MHz).

and append.

We have fully implemented backtracking, trailing, suspension/resumption of constraints/goals, deadlock detection and handling (*i.e.* labeling). We have also implemented decision graph compilation for the most commonly used ask constraints, joining of tail-recursive call, compilation of user constraints (any linear ask/tell inequality, equality and disequality constraint), DPAM code generator and native code generator for DECmpp. However, a number of less commonly used ask and tell constraints, borrowing and a garbage collector are yet to be implemented.

6.3 Constraints

Our current implementation compiles tell constraints directly to DECmpp native code. The compilation is very simple. While *clp(FD)* [DC93] has abstract machine instructions to install tell constraints, it is not needed in a fully concurrent system like Firebird. Since all tell constraints are regarded as processes, the same *call* instruction can be used for both goals and constraints.

As the result of compilation, we do not have to interpret tell constraints at runtime. We have employed a number of optimizations, like using a shift instruction for multiplying and dividing a constant which is a power of 2 (*i.e.* 2^i).

We shall compare and contrast the handling of equality tell constraints adopted by Firebird and that of CHIP [DVHS+88].

6.3.1 Breaking Down Equality Constraints

According to [VH89], A linear constraint of the form

$$a_1X_1 + \cdots + a_nX_n + c = b_1Y_1 + \cdots + b_mY_m$$

where $a_1, \ldots, a_n, b_1, \ldots, b_n$ are natural numbers and $X_1, \ldots, X_n, Y_1, \ldots, Y_m$ are d-variables with non-negative domains, is broken down into two constituents.

$$a_1X_1 + \cdots + a_nX_n + c = Z$$

$$b_1Y_1 + \cdots + b_mY_m = Z$$

Since the two constraints are similar, we shall consider $a_1X_1+\cdots+a_nX_n+c = Z$ only. We deduce that, for $1 \leq i \leq n$,

$$X_i = \frac{Z - \sum_{j \neq i} a_jX_j}{a_i}$$

Since all the coefficients and variables are positive, we can deduce the new upper and lower bounds of X_i given the upper and lower bounds of $X_j, j \neq i$. The division must be performed with inward rounding. The order which the X_i's should be handled is not stated clearly in [VH89], but from a commercially available CHIP system we know that each X_i is handled only once and finally Z is handled. Then the constraint is considered to be locally stable.

The reason why it works most of the time is as follows.

1. Z is handled after $X_i, 1 \leq i \leq n$, using the most updated upper and lower bounds of X_i.

2. As a result, any change in $X_i, 1 \leq i \leq n$ will cause a change in Z.

3. Any change induced by $a_1X_1 + \cdots + a_nX_n + c = Z$ to Z causes $b_1Y_1 + \cdots + b_mY_m = Z$ to be re-evaluated. Evaluating $b_1Y_1 + \cdots + b_mY_m = Z$ in turn causes Z to be changed, reinvoking $a_1X_1 + \cdots + a_nX_n + c = Z$. The two constraints are evaluated in a coroutine-like manner.

In general, this method does not give the smallest interval for all X_i. A counterexample is $5X_1 = 3Y_1 + 2Y_2$, $X_1 = 4$, $Y_1, Y_2 \in [0, 6]$. The constraint is broken down to $5X_1 = Z$ and $3Y_1 + 2Y_2 = Z$. Z is found to be 20. Y_1 is found to be $[3, 6]$. Next, Y_2 is found to be $[1, 5]$. However, if we handle Y_1 again, it can be further reduced to $[4, 6]$. From a commercially available CHIP system, we obtain $Y_1 \in [3, 6]$. The inconsistency is not discovered until Y_1 is labeled. However, if we repeat the constraint again (*i.e.* stating the same constraint twice in the query), we obtain $Y_1 \in [4, 6]$.

6.3.2 Processing the Constraint 'As Is'

In Firebird, a linear constraint

$$a_1X_1 + \cdots + a_nX_n + c = 0$$

is processed 'as is', without any decomposition. Furthermore, $a_1, \ldots, a_n$ and $X_1, \ldots, X_n$ are not required to be non-negative. For each X_i, the following formula is used to deduce the upper/lower bounds of X_i given the upper/lower bounds of each X_j, $j \neq i$.

$$X_i = \frac{c - \sum_{j \neq i} a_j X_j}{a_i}$$

The X_i's are processed in turn repeatedly until all of their upper and lower bounds become stable. Compared to CHIP, the pruning of invalid values is more complete.

6.4 The Wide-Tag Architecture

One of the main problems we encounter in our implementation is that only 64K bytes of memory is available for each processor element. To save memory, the *wide-tag memory architecture* is used. For 32-bit heap cells, only 14 bits are required to address the memory. These 14 bits are used to store the address of the first argument of a compound term and the remaining bits can be used to store the principal functor and arity. Figure 6.1 shows the heap cell format used in our DECmpp implementation. As a corollary, the differentiation between structure, list and constant becomes obsolete. A reference pointer becomes a compound term with functor and arity fields zeroed (Figure 6.2a), a constant is represented by a term with zero arity and no first argument pointer (Figure 6.2b) and a list becomes `./2` (Figure 6.2c). A compound term is shown in Figure 6.2d. Unlike WAM, an unbound variable is not a self-referential pointer but a reference pointer pointing to an unbound variable cell (see Figure 6.3), in which the head and the tail of a suspension list node are packed. Stack variables[4] are not possible and every variable must occupy at least a single heap cell.

[4]In WAM, a permanent variable which appears only in the body of a clause is stored as a single cell on the environment stack, consuming no heap space. This is called a *stack variable*.

reference	0's		pointer (14 bits)	00
structure	functor (11 bits)	arity (5 bits)	pointer (14 bits)	00
h-variable	head (16 bits)		tail (14 bits)	01
integer	value (30 bits)			10
d-variable	head (16 bits)		tail (14 bits)	11

Figure 6.1: Heap cell format

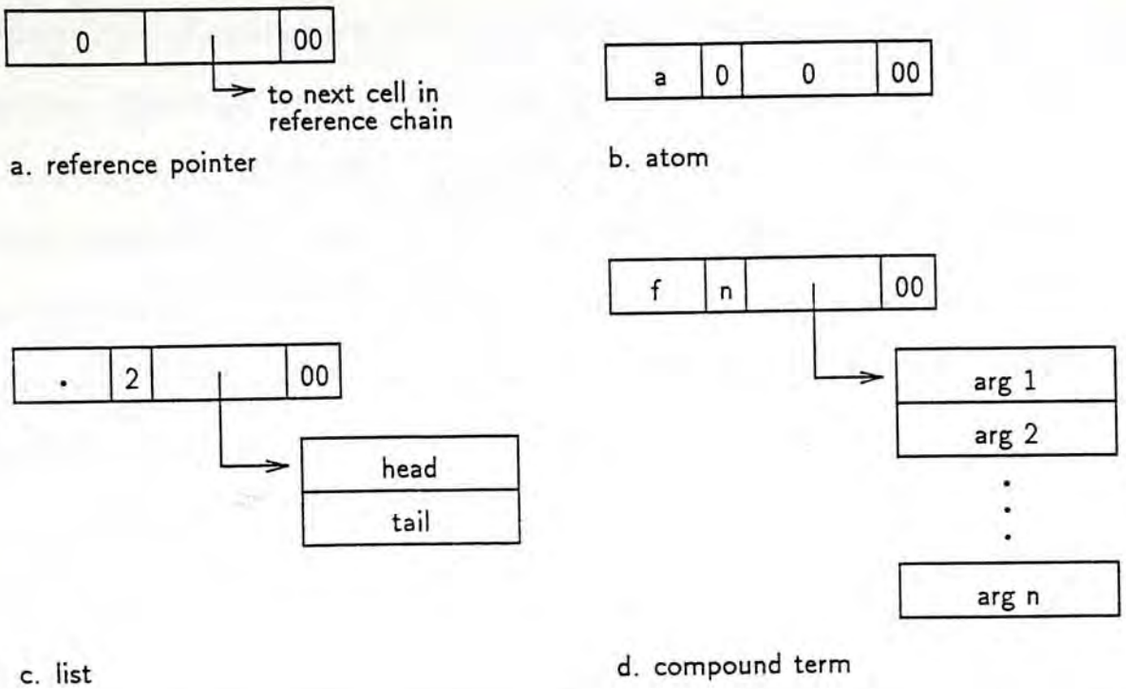


Figure 6.2: Examples: reference pointer, atom, list and compound term

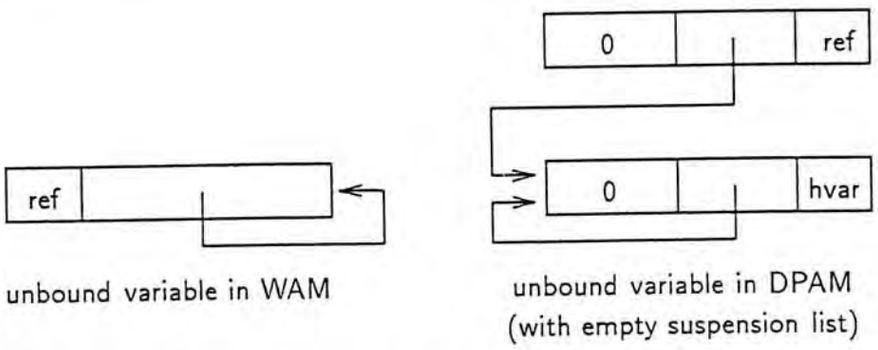


Figure 6.3: Unbound variable representations of WAM and DPAM

6.5 Register Window

The ACU has 32 32-bit scalar registers (called *CReg*), and each processor element has 40 32-bit vector registers (called *PReg*). A vector register is addressable by a scalar register plus an offset. *Register windows* are used to pass call arguments and store local variables, as follows. A scalar register is used as the *register*

pointer (`rp`). A procedure must not access any register whose number is lower than `rp`. The caller sets `rp` before calling another procedure and registers below `rp` are safe and will remain the same upon return. The caller has a *register frame pointer* (`rf`) to keep track of its own `rp`. The relationship between `rp` and `rf` is analogous to that between the stack pointer and the frame pointer in imperative languages (see Figure 6.4). Consequently an environment stack frame is not necessary under most circumstances.[5]

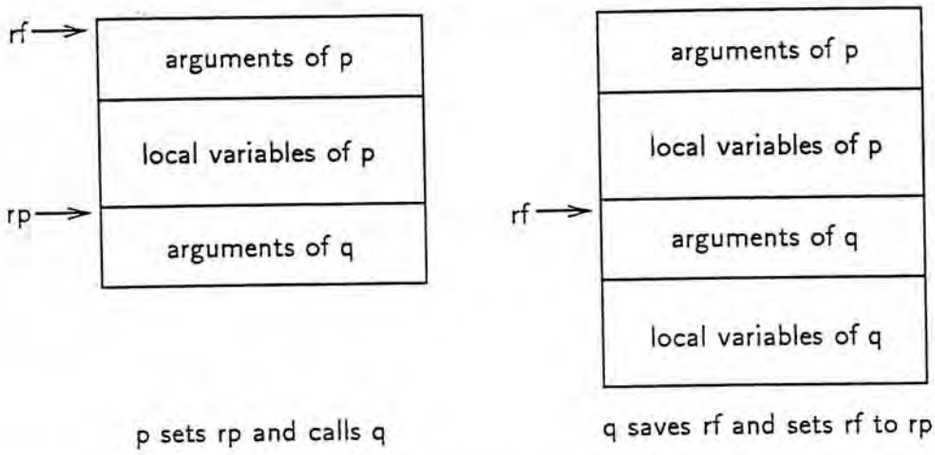


Figure 6.4: Register windows

6.6 Dereferencing

The dereferencing operation often introduces pipeline stalls in implementations[6]. The problem is worse on DECmpp since local indirect addressing is particularly

[5]In order to support register windows, we do not use the *MasPar Application Language* (MPL) [Chr90] to implement our system libraries but assembly language is used instead.

[6]Pipelining is very popular in modern microprocessor design. Memory operations frequently use different pipeline stages from other operations, resulting in empty (hence underutilized) pipeline stages. This is termed *a pipeline stall*. If one examines the disassembly of the dereferencing operation, she/he may find (depending on the particular compiler and architecture) that the data obtained from a memory fetch instruction is immediately used in the next instruction, leading to pipeline stalls.

slow. Therefore, we implemented a low-level operation called `deref_double` which performs the dereferencing of two terms simultaneously. While one thread is performing a memory load, the other is checking the tag. This operation is up to 30% faster than two separate dereferencing operations for two reference chains of equal length. The operation is used in many system library predicates (*e.g.* `in/2`, `is/2`, `=/2`).

6.7 Output

Displaying all the solutions on the front-end workstation of a massively parallel computer turns out to be the bottleneck of the system. To partially alleviate this, a *parallel reduction algorithm* has been implemented both to compress the solutions before they are transferred from the back-end to the front-end and to facilitate output. The idea is based on sequential backtracking.

6.7.1 Collecting the Solutions

The solutions from the data-parallel back-end are collected and stored in a scalar *output buffer*. If there is only one solution, the solution (which is a term, and hence a tree) is traversed in a depth-first, left-to-right manner and each visited node is copied to the output buffer. The result will be a prefix representation of the solution tree.

Consider the case with more than one solution. The tree is traversed as usual if the visited node is the same in all physical partitions. Otherwise, a choice point is created, a mark is written on the output buffer and an arbitrary alternative is taken. The mask is set to the set of physical partitions which take

the chosen alternative. Other alternatives will be taken upon backtracking. For example, if there are 5 partitions,

```
a(1,x).
a(1,y).
a(1,z).
a(2,x).
a(2,y).
```

We start at the root and write an `a/2` to the output buffer. Next the first argument is traversed and there are 2 alternatives. A mark is written on the output buffer and the alternative with `1` as the first argument is taken. The partitions with 2 as the first argument will stay idle until backtracking. At this point the output buffer will be

```
a/2  mark  1
```

The second argument is traversed and an additional choice point is created.

```
a/2  mark  1  mark  x/0
```

Upon backtracking, the other alternatives (either `y/0` or `z/0`) are taken. Suppose `y/0` is taken. A mark is written on the output buffer, followed by `y/0`.

```
a/2  mark  1  mark  x/0  mark  y/0
```

Execution continues and the final output buffer is

```
a/2  mark  1  mark  x/0  mark  y/0  z/0  2  mark  x/0  y/0
```

A combined *environment/choice point stack*, similar to that of WAM [AK91], is employed to maintain the choice points and traverse the tree. The stack has both scalar and vector components. The specific implementation details are out of the scope of this thesis. If the solution contains unbound domain variables, they are labeled on the fly.

6.7.2 Decoding the solution

To display the solutions on the front-end workstation, another combined *environment/choice point stack* and a *text buffer* is maintained. The pointer to the top of the text buffer, called *text pointer* is saved in each choice point so that the first part of the text buffer can be reused. Using the example in the last section (Section 6.7.1), after `a/2` is processed the following is written on the text buffer:

```
a(
```

Since a mark follows, a choice point is created and the text pointer is saved. Execution continues and another mark is encountered.

```
a(1,
```

As a result, another choice point is created. Finally, the first solution is formed and displayed.

```
a(1,x).
```

Backtracking is needed and the text pointer is restored.

```
a(1,
```

The word `y/0` in the output buffer is processed.

```
a(1,y).
```

In a similar manner the other solutions are displayed. Our algorithm can also handle operators and lists, but once again, such implementation details are out of the scope of this thesis.

Chapter 7

Performance

The aim of this chapter is to evaluate the performance of our data-parallel implementation and to analyze the effects of a number of design decisions.

The reader may find the following performance results unsatisfactory. While one would expect a linear speed up of 8,192 on 8,192-processor elements, we attain a maximum speedup of only 121 on the 9-queens problem. The speedup drops to 20.3 when the performance of a 8,192-processor DECmpp is compared to CHIP on DECstation 3100, and the speedup figure will look even less impressive when compared to say a sequential implementation on DEC Alpha AXP. However, the reader is reminded that:

1. Firebird is not optimized for linear speedup but for extremely high degrees of parallelism.

2. We use a machine with 8,192 tiny, slow 4-bit engines using the accumulator architecture. 16 such tiny processor elements form a cluster and share a single 8-bit memory port. Our implementation platform is inherently slow. Therefore, poor performance is not necessarily an indication of poor design.

We are also unable to test very large problems due to the limited amount of local memory available on each processor element (64K bytes). Again, this is not a restriction imposed by Firebird or DPAM. Although there is always room for improvement in Firebird, the reader is requested to distinguish the limitations of a particular implementation platform from the limitations of our data-parallel execution model.

In all of the following benchmarks, t_1, t_2, *etc* are execution times for all solutions in seconds, $\#proc$ is the number of processor elements, DL the number of deadlocks, BT the number of backtrackings and P the number of partitions. P may change in the course of execution, but it can never exceed the number of processor elements. Only the value of P taken at the end of execution is given. A dash indicates that a benchmark is not available because memory is not enough for its execution. Since many parameters are studied, only one parameter is varied in each benchmark. The other parameters are listed below each table. The benchmark set is shown in Table 7.1.

`send`	SEND + MORE = MONEY
`eq10`	10 simultaneous linear equations over 7 variables
`eq20`	20 simultaneous linear equations over 7 variables
`queen`	n-queens problem
`magic`	magic series problem
`magich`	magic series problem with redundant constraint $\sum_{i=0}^{n} s_i = n$.

Table 7.1: Benchmark set

7.1 Uniprocessor Performance

We measure the performance of our implementation on a DECmpp 12000 Sx-100 massively parallel computer, but using only one of the processor elements. The result is compared with CHIP version 3.2 on a DECstation 3100. The CHIP benchmark is obtained using generalized forward checking but no first fail heuristics. The Firebird system is modified to apply nondeterministic derivation (*i.e.* labeling) to the domain variables in the order they are created. These ensure that the order of labeling of CHIP is the same as Firebird. The time used to find all solutions in seconds, neglecting any time used for input/output, is given in Table 7.2.

Except for the magic series problem, the results indicate that our implementation has very poor performance compared to CHIP. The magic series problem has good performance because Firebird is a concurrent language which allows a different formulation of the problem (see Section 2.5.1.3).

We attribute this to the deficiency of our implementation platform. We profile our execution and find that an average machine instruction requires about 10 machine cycles to execute on our 12.5 MHz DECmpp 12000 Sx-100. In general, most sequential instructions require only 1 machine cycle to execute, while some parallel instructions take several hundred cycles. Table 7.3 lists the average execution time, in machine cycles, of a number of machine instructions. The operands of the instructions are registers. Each machine instruction in DECmpp has only 2 operands. Sometimes two 2-operand instructions are required to do the same task as a single 3-operand instruction typical of modern RISC architectures. Furthermore, we found that some other instruction sequences can be

benchmark	Firebird			CHIP	
	t_1	DL	BT	t_2	t_2/t_1
`send`	.028	2	3	.007	.25
`eq10`	4.720	21	163	.660	.14
`eq20`	5.153	13	115	.848	.16
`queen(4)`	.028	3	5	.007	.25
`queen(6)`	.239	27	39	.047	.20
`queen(8)`	3.172	265	415	.512	.16
`queen(10)`	60.002	4229	6665	8.547	.14
`queen(12)`	-	-	-	211.948	-
`magic(3)`	.110	2	7	.051	.46
`magic(6)`	2.128	5	31	6.378	3.00
`magic(8)`	8.634	7	57	212.303	24.59
`magich(3)`	.127	2	6	.058	.46
`magich(6)`	1.112	4	19	2.422	2.18
`magich(9)`	4.747	7	41	140.886	29.68

Test conditions: $\#proc$=1, eager bit vector creation, eager nondeterministic derivation, no solitary memory access, no priority scheduling.

Table 7.2: Benchmark: uniprocessor performance

Parallel instructions				Sequential instructions	
mov32	18.9	add32	24.9	cmov32	1.0
ld32	79.0	mul32	244.6	cld32	5.2
ld32 (indirect)	224.6	div32	422.9	cst32	6.3
ldsol32	18.6	mod32	469.3	cadd32	1.0
ldsol32 (indirect)	43.5	shll32	63.9	cjmp	3.2

Table 7.3: Execution time of machine instructions (in machine cycles)

replaced by a single instruction on a RISC computer[1]. For these reasons, we estimate that a single processor of our implementation platform is about 50 times slower than an average workstation. The same result has been observed in the implementation of SIMD MultiLog [Smi93].

One could always enhance the speed of individual processor elements to get around this problem. We tested an earlier version of Firebird on a newer model, MasPar MP-2, which was binary compatible with the DECmpp 12000 Sx-100 we are currently using, and found that it was about 2 times faster on the n-queens problem.[2] See [TL93] for the benchmark.

7.2 Solitary Mode

On our implementation platform, DECmpp, 16 processor elements form a cluster and share a single 8-bit memory port. We would prefer each processor element to have its own 32-bit memory port, giving 64 times of memory bandwidth, in order to be competitive with a workstation. Although we used the overlapping feature so that memory access can be performed in parallel with other machine instructions whenever possible, a concurrent constraint programming system is so memory intensive that performance is completely dominated by memory access time.

We introduced a *solitary mode* to alleviate this memory bottleneck. Only

[1]For example, a conditional branch. One instruction is used to move each processor's flag to the contingent bit. The next instruction obtains the global or-ing of all contingent bits and stores the result in the carry flag of the array control unit. The last instruction is the actual branch.

[2]The manufacturer claimed a speedup of up to 4.5, without any modifications to the program. The exact reason why we obtained only a speedup of 2 was unknown, but we suspected that this was because memory throughput was made only 2 times faster, and our system was very memory intensive.

1 processor element in each cluster is used, so that the processor element has exclusive access to the memory port. Each `ld` (load) and `st` (store) instruction is replaced by the corresponding solitary equivalent (`ldsol` and `stsol`). The performance of 512 processor elements in solitary mode is compared with 8,192 processor elements in normal mode. See Table 7.4.

	solitary (#*proc*=512)				normal (#*proc*=8,192)				
benchmark	t_1	*DL*	*P*	*BT*	t_2	*DL*	*P*	*BT*	t_2/t_1
`send`	.014	2	4	0	.019	2	4	0	1.36
`eq10`	.701	6	76	28	.334	6	76	9	.48
`eq20`	.757	3	58	10	.519	3	140	0	.69
`queen(4)`	.009	2	6	0	.014	2	6	0	1.56
`queen(6)`	.028	4	40	0	.042	4	40	0	1.50
`queen(8)`	.157	11	182	4	.101	7	548	0	.64
`queen(10)`	3.957	133	366	191	1.400	27	2399	17	.35
`queen(12)`	-	-	-	-	92.183	971	4165	1191	-
`magic(3)`	.051	2	11	0	.074	2	11	0	1.45
`magic(6)`	.514	2	38	0	.791	2	38	0	1.54
`magic(9)`	1.957	2	83	0	2.909	2	83	0	1.49
`magich(3)`	.060	2	7	0	.086	2	7	0	1.43
`magich(6)`	.282	2	21	0	.411	2	21	0	1.46
`magich(9)`	.833	2	45	0	1.210	2	45	0	1.45
`magich(12)`	1.787	2	78	0	2.544	2	78	0	1.42

Test conditions: eager bit vector creation, eager nondeterministic derivation, no priority scheduling.

Table 7.4: Benchmark: solitary mode performance

From Table 7.3, a solitary memory instruction is 4–5 times faster. By improving the speed of memory access alone a speedup of 1.5 is attained (despite

some memory instructions are executed in background). The advantage of solitary mode is lost when 512 processor elements are not enough and backtracking is required, as in `eq10`, `eq20`, `queen(8)` and `queen(10)`.

7.3 Bit Vectors of Domain Variables

The domain of a domain variable is represented by a bit vector. Many newer finite domain constraint programming systems, like *clp(FD)* [DC93] and *cc(FD)* [VHSD93], does not have bit vectors for continuous domains. A bit vector is created on demand only when the domain is broken into two parts because one of the invalid values is removed. For example, if $X \in \{1 \ldots 5\}, X \neq 1$, then $X \in \{2 \ldots 5\}$ and a bit vector is unnecessary. However, if $X \in \{1 \ldots 5\}, X \neq 3$, then $X \in \{1, 2, 4, 5\}$ and a bit vector representing the domain is created. We test the effect of this optimization in a data-parallel context.

The optimization leads to a slight reduction of both heap consumption and execution time, except for the n-queens problem, where both execution time and memory consumption are made worse. We find that several bit vectors may be created for a single domain variable. For example, suppose $X \in \{1 \ldots 5\}$ and $X \neq Y$, where Y is 1 in partition 1, 3 in partition 2 and 5 in partition 3. As a result, a bit vector is created for the X in partition 2 only. If there is another constraint $X \neq Z$, where Z is 3 in partition 1, 4 in partition 2 and 5 in partition 3, a bit vector will be created for partition 1. Two bit vectors have been created, leading to slower execution. Under the heap frame scheme both bit vectors consume heap memory.

We devise an *eager creation* scheme to get around this problem. Note that

a bit vector can be created only when a disequality constraint is encountered. In processing a disequality constraint, if any of the physical partitions needs a bit vector, bit vectors will be created for all physical partitions. *Lazy creation* refers to the scheme in which bit vectors are created only for partitions in need. The 3 schemes are compared in Table 7.5. The heap and trail usages are given in bytes.

The eager creation scheme is slightly better than the unoptimized version on average, and is preferred because sometimes very large continuous domains may appear in users' programs. The eager bit vector creation scheme has been used to obtain the performance results in previous sections.

	eager creation			lazy creation			unoptimized		
benchmark	t_1	heap	trail	t_2	heap	trail	t_3	heap	trail
send	.019	460	0	.019	460	0	.019	472	0
eq10	.334	1728	162	.334	1728	162	.345	1756	162
eq20	.519	3636	0	.519	3636	0	.528	3664	0
queen(4)	.014	440	0	.014	452	0	.013	444	0
queen(6)	.042	964	0	.042	996	0	.042	968	0
queen(8)	.101	1656	0	.101	1708	0	.101	1660	0
queen(10)	1.400	3208	780	1.400	3276	780	1.397	3212	780
queen(12)	92.183	5404	1764	92.177	5484	1764	92.062	5408	1764
magic(3)	.074	2044	0	.074	2044	0	.077	2252	0
magic(6)	.791	13512	0	.791	13512	0	.822	14100	0
magic(9)	2.909	41476	0	2.908	41480	0	3.006	42644	0
magich(3)	.086	2324	0	.086	2324	0	.090	2512	0
magich(6)	.411	9276	0	.411	9276	0	.429	9800	0
magich(9)	1.210	24280	0	1.210	24280	0	1.260	25368	0
magich(12)	2.544	47596	0	2.543	47596	0	2.644	49432	0

Test conditions: $\#proc$=8,192, eager nondeterministic derivation, no solitary memory access, no priority scheduling.

Table 7.5: Benchmark: on demand creation of bit vectors

7.4 Heap Consumption of the Heap Frame Scheme

The heap frame scheme (Section 5.1.3.3, page 38) is aimed at improving memory access time when building heap terms. However, it has the drawback that sometimes heap fragmentation occurs. Let H_1 be the heap consumption (in bytes) when the heap frame scheme is used, and H_2 be the heap consumption when the heap frame scheme is not used. We define *percentage fragmentation* as

$$\frac{H_1 - H_2}{H_1} \times 100\%$$

Our results indicate that fragmentation occurs only in the magic series problem (Table 7.6). For all the other programs in our benchmark set, fragmentation is zero.

	`magic`			`magich`			
n	3	6	9	3	6	9	12
frag (%)	1.96	5.98	7.69	1.03	1.21	5.27	6.19

Test conditions: $\#proc$=8,192, eager bit vector creation, eager nondeterministic derivation, no solitary memory access, no priority scheduling.

Table 7.6: Benchmark: heap fragmentation

From Table 7.3 (page 72), we find that a direct addressing 32-bit load instruction (`ld32`) is 2.8 times faster than its local indirect addressing counterpart. We have also shown in Section 7.2 (page 73) that memory access time has a great impact on system performance. Therefore, we believe that the slight memory overhead of the heap frame scheme is acceptable.

7.5 Eager Nondeterministic Derivation vs Lazy Nondeterministic Derivation

As discussed in Section 4.4, it is not necessary or even desirable to wait for a deadlock before a nondeterministic derivation step is applied. With *eager nondeterministic derivation*, a labeling process is moved to the ready queue whenever

	lazy nondet. derivation				eager nondet. derivation				
benchmark	t_1	DL	P	BT	t_2	DL	P	BT	t_2/t_1
send	.019	2	4	0	.019	2	4	0	1.00
eq10	.537	4	164	0	.334	4	181	8	.62
eq20	.791	3	116	0	.519	3	140	0	.66
queen(4)	.016	2	6	0	.014	2	6	0	.88
queen(6)	.054	4	40	0	.042	4	40	0	.78
queen(8)	.201	7	416	0	.101	7	548	0	.50
queen(10)	3.542	27	2397	11	1.400	27	2399	17	.40
queen(12)	209.056	952	4171	1027	92.183	971	4165	1191	.44
magic(3)	.082	2	8	0	.074	2	11	0	.90
magic(6)	.814	2	32	0	.791	2	38	0	.97
magic(9)	3.379	2	74	0	2.909	2	83	0	.86
magich(3)	.086	2	7	0	.086	2	7	0	1.00
magich(6)	.425	2	20	0	.411	2	21	0	.97
magich(9)	1.284	2	42	0	1.210	2	45	0	.94
magich(12)	2.721	2	74	0	2.544	2	78	0	.93

Test conditions: $\#proc$=8,192, eager bit vector creation, no solitary memory access, no priority scheduling.

Table 7.7: Benchmark: lazy nondeterministic derivation vs eager nondeterministic derivation

deadlock of any physical partition is detected. *Lazy nondeterministic derivation* refers to the control strategy in which a labeling process is moved to the ready queue only after the deadlock of all physical partitions is detected. *Eager nondeterministic derivation* is chosen over *lazy nondeterministic derivation* for DPAM and we justify this choice using empirical results (Table 7.7).

From the results, we find that lazy/eager nondeterministic derivation is basically an execution time/degree of parallelism tradeoff. Eager nondeterministic derivation creates partitions more aggressively. As a result more parallelism can be exploited, leading to better performance. It is worth noting that although lazy nondeterministic derivation reduces the number of backtrackings, it does not lead to any performance gain.

7.6 Priority Scheduling

One of the ways to increase the degree of parallelism is to schedule those processes resumed by a labeling process ahead of all the others. We have implemented a prototype of this *priority scheduling* scheme on top of our system and its effect on performance is measured (Table 7.8).

Priority scheduling is faster on average, with best performance on the n-queens problem. A possible explanation is that priority scheduling is more suitable when disequality constraints are predominant.

benchmark	priority scheduling				w/o priority scheduling				t_2/t_1
	t_1	DL	P	BT	t_2	DL	P	BT	
`send`	.019	2	4	0	.019	2	4	0	1.00
`eq10`	.299	4	226	0	.334	4	181	8	1.12
`eq20`	.429	3	138	0	.519	3	140	0	1.21
`queen(4)`	.013	2	6	0	.014	2	6	0	1.08
`queen(6)`	.034	4	46	0	.042	4	40	0	1.24
`queen(8)`	.075	7	564	0	.101	7	548	0	1.35
`queen(10)`	.827	27	2530	15	1.400	27	2399	17	1.69
`queen(12)`	57.264	952	4178	1027	92.183	971	4165	1191	1.61
`magic(3)`	.076	2	8	0	.074	2	11	0	.97
`magic(6)`	.698	2	38	0	.791	2	38	0	1.13
`magic(9)`	3.204	2	83	0	2.909	2	83	0	.91
`magich(3)`	.090	2	7	0	.086	2	7	0	.96
`magich(6)`	.463	2	22	0	.411	2	21	0	.89
`magich(9)`	1.315	2	45	0	1.210	2	45	0	.92
`magich(12)`	2.729	2	78	0	2.544	2	78	0	.93

Test conditions: $\#proc$=8,192, eager bit vector creation, eager nondeterministic derivation, no solitary memory access.

Table 7.8: Benchmark: priority scheduling

7.7 Execution Profile

We measure the time spent in nondeterministic derivation (ND), constraint solving (C), backtracking (BT) and the execution time up to the first deadlock (FDL). The backtracking time includes the time to update the choice point, unwind the trail and restore the process queues. The percentage time is shown in Table 7.9.

benchmark	*FDL*	*C*	*ND*	*BT*
`send`	75.0	78.0	2.5	0
`eq10`	7.0	95.8	.7	1.9
`eq20`	16.2	98.1	.9	0
`queen(4)`	52.3	59.7	.06	0
`queen(6)`	35.8	72.7	.06	0
`queen(8)`	25.8	80.6	.05	0
`queen(10)`	2.9	91.3	3.4	2.7
`queen(12)`	.08	90.9	2.8	3.7
`magic(3)`	40.6	57.1	.9	0
`magic(6)`	11.0	63.2	.1	0
`magic(9)`	6.0	60.4	.04	0
`magich(3)`	36.5	61.2	.9	0
`magich(6)`	19.7	66.6	.3	0
`magich(9)`	14.5	68.3	.1	0
`magich(12)`	11.3	68.5	.06	0

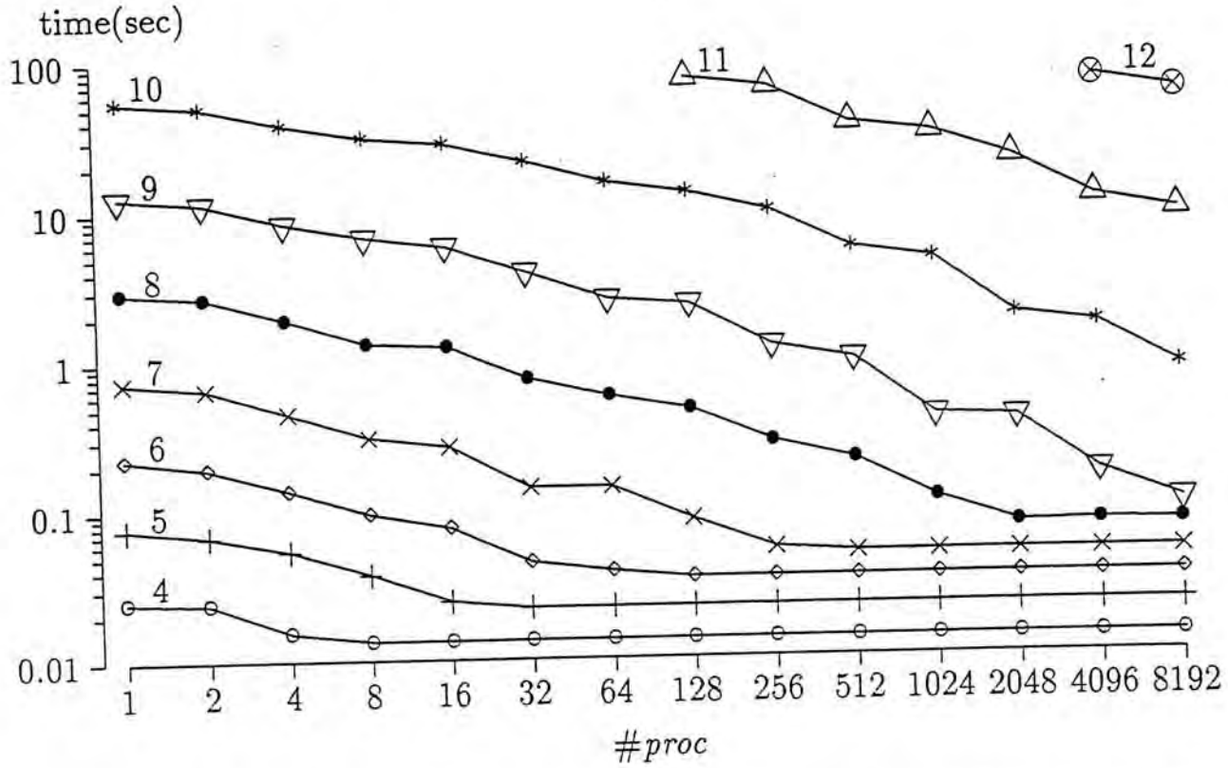
Test conditions: *#proc*=8,192, eager bit vector creation, eager nondeterministic derivation, no solitary memory access, no priority scheduling.

Table 7.9: Benchmark: execution profile (in %)

It is evident that constraint solving dominates execution time for large problems. This implies that compiler optimization may not be as useful as an efficient constraint solver. The magic series problem uses less time to solve constraints than other programs because a fair amount of time is spent in the `bool/3` predicate (refer to Section 2.5.1.3, page 13). Nondeterministic derivation and backtracking overhead is almost negligible.

7.8 Effect of the Number of Processor Elements on Performance

Figure 7.1 shows the execution time of n-queens, for $4 \leq n \leq 12$, with 1 to 8,192 processor elements. We use priority scheduling because it is particularly suitable for the n-queens problem.



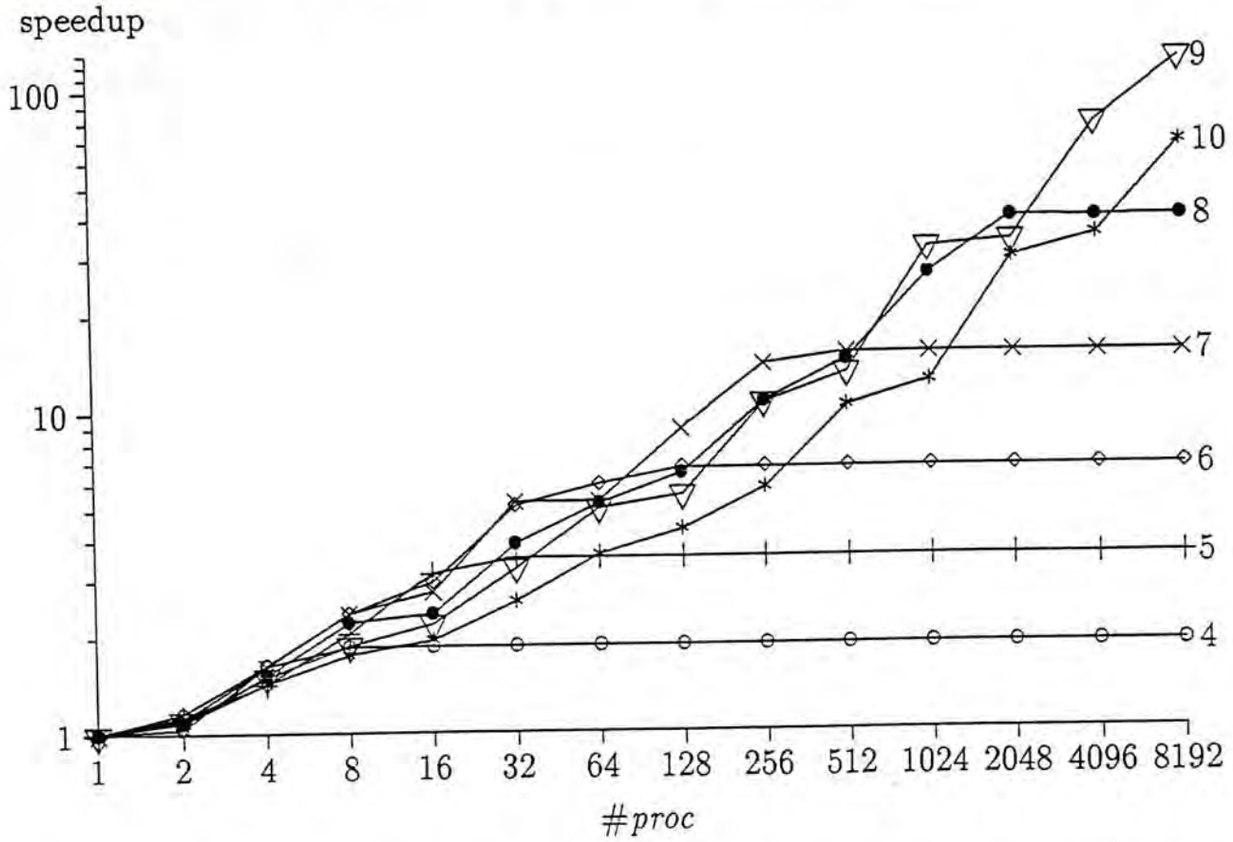
Test conditions: eager bit vector creation, eager nondeterministic derivation, no solitary memory access, priority scheduling.

Figure 7.1: Benchmark: run time of n-queens program

We do not have enough memory (each processor element has only 64K bytes) to run 13-queens. Some data points are missing from the above graph for the same reason. A lot of memory is consumed if backtracking is used because process structures created before the last choice point cannot be freed until

backtracking occurs.

Figure 7.2 shows the speed up with respect to the execution time using 1 processor element as the number of processor elements is varied.



Test conditions: eager bit vector creation, eager nondeterministic derivation, no solitary memory access, priority scheduling.

Figure 7.2: Benchmark: speedup of N-Queens Program

The speed up value continues to increase until there are too many processor elements to be utilized. Otherwise, the speed up value is fairly independent on problem size and fairly constant for a given number of processor elements. Speed up is scalable but sublinear. For instance, the speed up of 7-queens levels at about 15, but 512 processor elements are required to obtain this speed up. The reason for this is that the processor elements are divided evenly in

a nondeterministic derivation step, which may not be the optimal processor allocation strategy. Furthermore, a processor element will be remain idle after failure until the system backtracks.

We obtain a maximum speed up of 121 for 9-queens. We have only analyzed the performance of n-queens in detail, and the reader is reminded that not all other programs exhibit the same behaviour as n-queens.

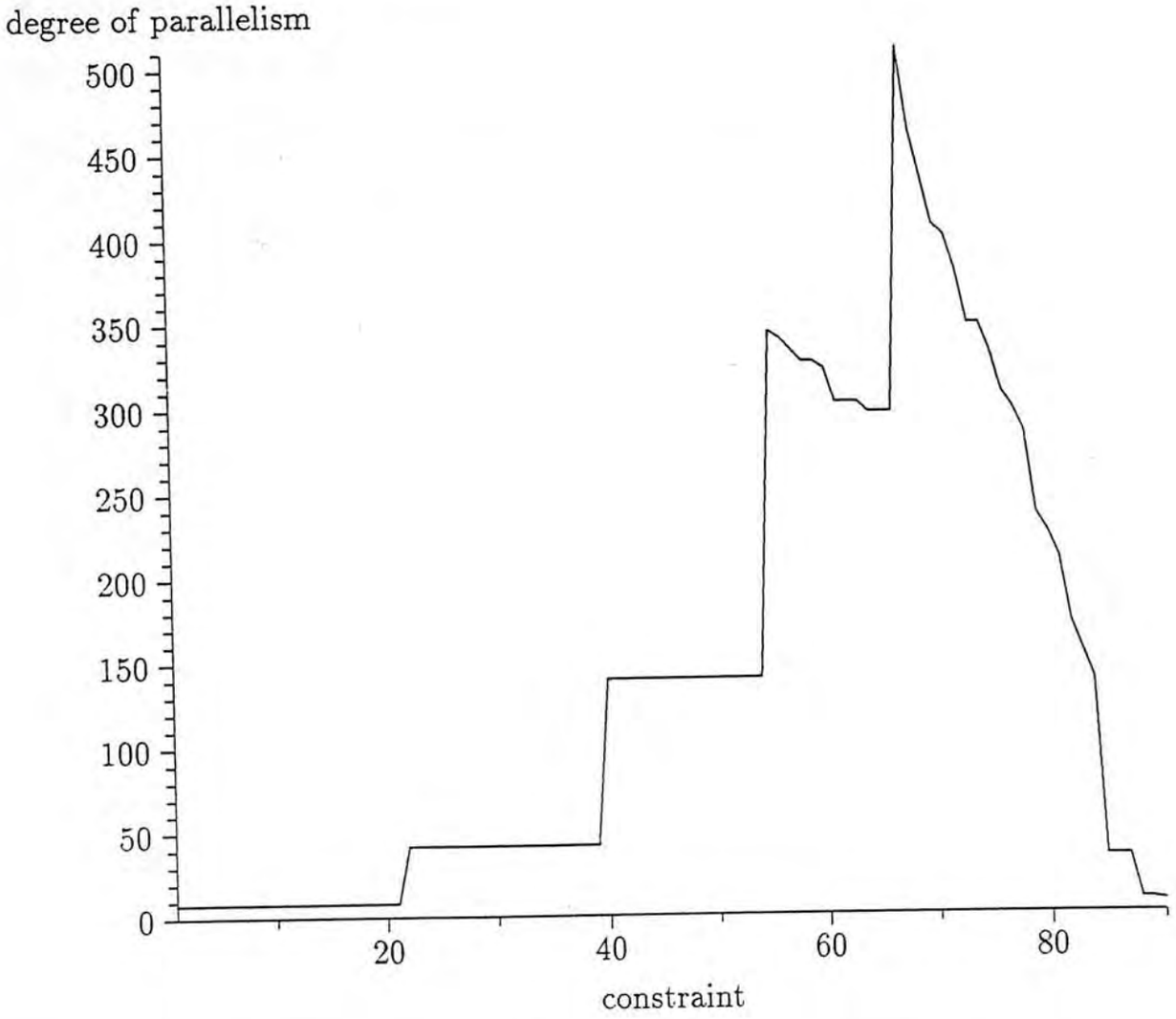
7.9 Change of the Degree of Parallelism During Execution

We are interested in the change of the degree of parallelism during execution. We count the number of active partitions when each constraint is executed and plot the graph in Figure 7.3. We do not include anything before the first nondeterministic derivation step, with the understanding that

1. there can only be a single active partition, and
2. execution time before the first nondeterministic derivation is only a small portion of the total execution time.

Furthermore, since we have found out that priority scheduling can improve the performance of the system, we use it when obtaining the plot. Other parameters remain unchanged.

From the plot we can identify the nondeterministic derivation steps as sudden leaps in the degree of parallelism. A peak of 510 is attained. After that, the degree of parallelism drops because of the failure of some partitions. We are actually approaching the theoretical limit of or-parallelism. Using or-parallelism



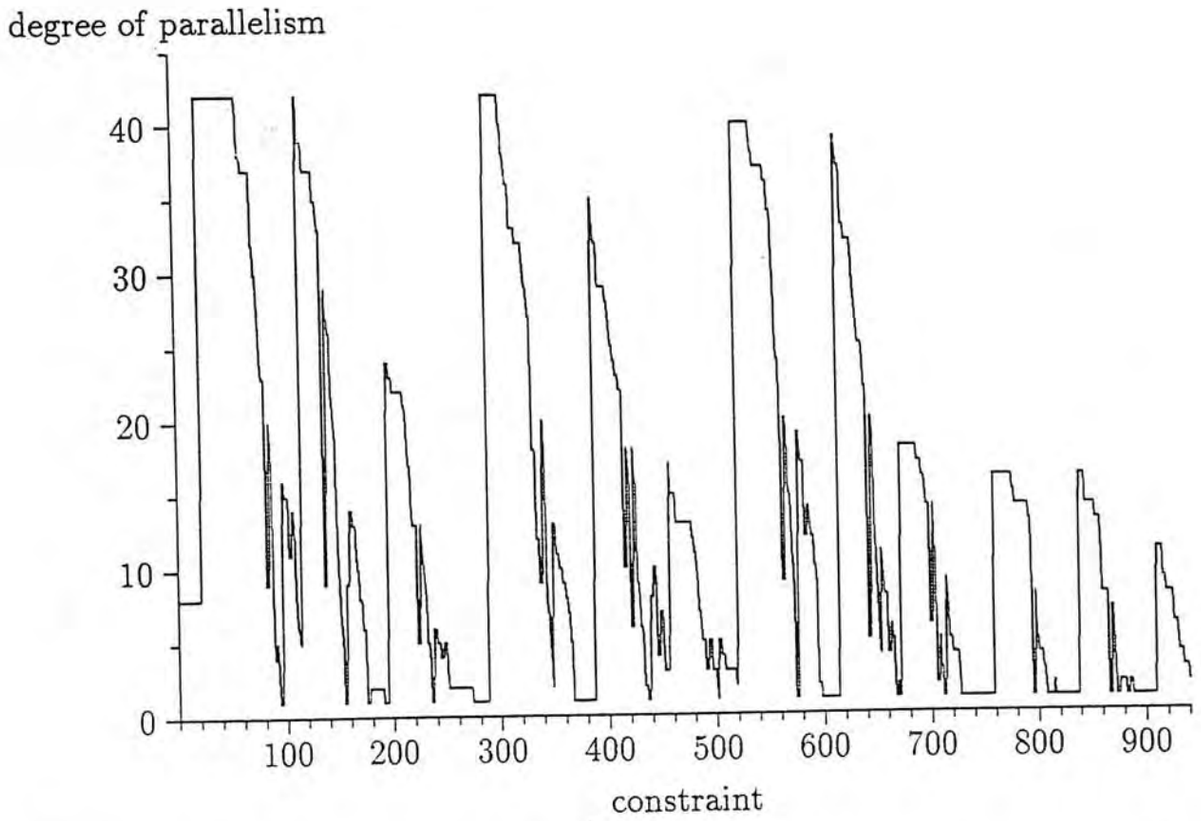
Test conditions: $\#proc$=8,192, eager bit vector creation, eager nondeterministic derivation, no solitary memory access, priority scheduling.

Figure 7.3: Execution trace of 8-queens, $\#proc$=8,192

alone the peak of 510 can never be exceeded, although the more flexible MIMD architecture may be able to exploit higher degrees of parallelism after that peak.

We conclude that the inherent limitation of or-parallelism will show up in any massively parallel implementation. Degree of parallelism rises slowly at the beginning, making full utilization of processor elements impossible. After the peak is reached, some or-branches fail, again limiting the degree of parallelism and hence processor element utilization.

Next, we show that when the number of processor elements is very small when compared to the number of or-branches, reasonably high processor element utilization can be maintained. For the 8-queens problem, 64 processor elements is just enough for the first two nondeterministic derivation steps. The plot is shown in Figure 7.4.



Test conditions: $\#proc$=64, eager bit vector creation, eager nondeterministic derivation, no solitary memory access, priority scheduling.

Figure 7.4: Execution trace of 8-queens, $\#proc$=64

Chapter 8

Related Work

Firebird can be regarded as flat GHC [Ued85] extended with finite domain constraint handling capabilities. The finite domain constraints used in Firebird originates from CHIP [VH89]. The semantics of Firebird is based on that of the algorithmic programming language ALPS [Mah87]. The concepts of ALPS can be applied to a large class of programs which arise in practice. However, it is not applicable to certain domains, including the finite domain, where the notion of nondeterminism is inherent. Nondeterministic derivation in Firebird is inspired by the Andorra Model [War90], but the operational semantics are fundamentally different.

In the early stage, we built a prototype implementation of a concurrent constraint logic programming language called *FD-Parlog* [LTC93], which is similar to Firebird but is based on *Parlog* [CG86].

Although DPAM does not look like WAM [AK91], many concepts like choice point and trailing are borrowed from WAM. The concurrent process scheduler

is based on JAM [Cra90b] (the abstract machine of Parallel Parlog). The implementation of Firebird is also influenced by *BinProlog* [Tar92a, Tar92b]. For instance, the wide-tag memory architecture is a reminiscence of the *tag on data* representation. Both schemes make the differentiation between lists, structures and atoms unnecessary. They are also more suitable for decision graph compilation [KS90] than WAM [AK91].

We are unaware of any parallel execution schemes for concurrent constraint programming or constraint logic programming languages on massively parallel SIMD computers. However, there have been a number of implementation schemes of logic programming languages on SIMD computers. [KB87, NT88a, NT88b, IIK90, BM92, BP92, Smi93].

8.1 Vectorization of Prolog

[KKS88] is probably the first or-parallel logic programming system on SIMD computers. The n-queens program is manually vectorized and executed on a Cray-type vector supercomputer, exploiting or-parallelism. Automatic vectorization is difficult. We address this problem by starting with a finite domain constraint language, where the vectorization is inherent in the nondeterministic derivation step (*i.e.* the labeling operation). Unlike supercomputers, the memory of DECmpp is distributed but we introduce a processor element allocation strategy which eliminates the need for interprocessor communication. [KS89] is the parallel backtracking scheme of [KKS88], but it has not been implemented.

8.2 Parallel Clause Matching

DAP Prolog [KB87] is an implementation of Prolog on DAP (*Distributed Array Processor*), a SIMD machine with 1,024 processing elements, each having a local memory, with a mesh-style interconnection network. When a clause head is matched against several clauses in the program, the unifications may be done in parallel. No execution time figures are given. Because the unifications are synchronized to begin simultaneously, sometimes large 'holes' may result and severely limit the utilization of processor elements when all the other unifications have finished, waiting for a very long one. Unfortunately, Firebird is susceptible to a similar problem.

Firebird executes different possible values in a domain in parallel. In parallel clause matching, the program is distributed over the processor elements, and a goal is matched against each clause in parallel. This precludes the possibility of compilation. In Firebird, programs are compiled to DPAM code before being interpreted or further translated to native code.

8.3 Parallel Interpreter

Theoretically, it is possible to emulate a MIMD machine using a SIMD machine. [NTS8a] took this approach to implement Fleng, a simplified version of flat GHC, on the Hitachi S-820 supercomputer. They used vector instructions to execute 256 processes in parallel. The operations are not synchronized. Instead, non-real-time operations are divided into small, real time steps. In their implementation, there is a queue for each kind of operation. A single step yields two queues, one for those processes which have not finished to be fed back for

the same operation again, and the other for those finished processes to proceed to the next operation. Consequently, high processor element utilization, high parallelism and near-linear speedup can be achieved. The inference engine can attain 1.1 MLIPS.

They have also implemented their system on a Connection Machine [NT88b], attaining a peak performance of about 108 KLIPS on a simulator. An alternate version based on μWAM, a simplified version of WAM achieved a maximum of about 392 KLIPS on a simulator. Their low performance can be attributed to both the overhead of interpretation and the extremely slow execution of each individual processor element on the Connection Machine (65,536 single-bit processor elements at a slow clock rate).

8.4 Bounded Quantifications

Barklund and Millroth [BM92] transformed recursive programs to iterative programs for parallel execution. Their sequential version attained a speedup of 15, but the implementation on Connection Machine model 200 with 4,096 processor elements is only about 2 times faster than the sequential version (*i.e.* the total speed up is about 30) [ABB92]. Again, it can be attributed to the slowness of each individual processor element on the Connection Machine.

8.5 SIMD MultiLog

SIMD MultiLog [Smi93] is another or-parallel system implemented on MasPar MP-1. A new *disj* operator is introduced. All solutions to goal G are collected

when *disj* G is used. The solutions form a disjunctive set of environments and goals appearing after G can be executed in these environments in parallel.

[KKS88], SIMD MultiLog and Firebird all execute goals over a disjunctive set of environments, exploiting or-parallelism. [KKS88] relies on a vectorizing compiler, MultiLog uses solution aggregation and in Firebird the environments fall out of the labeling operation on domain variables naturally.

Like our approach, MultiLog has the advantage that traditional compilation techniques are applicable. Furthermore, *engine* variables which reside on the host computer is distinguished manually from *multi* variables which reside on the processor elements. This leads to higher time and space efficiency. Automatic compilation of the engine/multi distinction is expected to be possible, and we are looking forward to incorporate this feature to future versions of Firebird. However, SIMD MultiLog has the overhead of environment copying which is not necessary in Firebird. [Smi93] points out that environment copying is a bottleneck of SIMD MultiLog. Another drawback of SIMD MultiLog is that processor element utilization is limited when backtracking is used. Backtracking is not dynamically invoked but must be tuned by the user before execution.

The maximum speedup attained with a 8,192-processor machine over a single processor element of the same machine is used is 1872.7 on the 20 bits palindrome problem with naïve reverse. However, 11 queens (Bratko)[1] attains only a speedup of 2.5 and an execution time of 18.7 seconds.

[1]This benchmark is taken from *Prolog Programming for Artificial Intelligence* by I. Bratko [Bra90]

Chapter 9

Conclusion

We have proposed a new concurrent constraint logic programming language called Firebird and its execution model on data-parallel computers. Firebird supports both concurrency and data-parallelism. Concurrency arises from the stream and-parallelism of ordinary committed-choice logic programming languages and can be exploited on shared-memory architectures. In a nondeterministic derivation step, one of the domain variables is selected and labeled, and each possible value in its domain is attempted in an or-parallel fashion. Data-parallelism is exploited in the resulting or-parallel execution. The Data-Parallel Abstract Machine is designed to implement this model. A subset of Firebird has been implemented on a DECmpp massively parallel computer and we have given some performance results.

We shall investigate the limitations of Firebird and suggest possible future work.

9.1 Limitations

9.1.1 Data-Parallel Firebird is Specialized

Just like a vectorizing Fortran compiler is targeted towards numerical problems, the data-parallel execution model of Firebird is aimed towards constraint satisfaction problems. In our data-parallel model, only or-parallelism is possible. For this reason, no flat GHC program without constraints can yield any speed up. Furthermore, although clause-based or-parallelism can be emulated by domain-variable-based or-parallelism, it's not efficient in a data-parallel setting. Consider the following example.

```
p(X) :- X = 0 | a, b.
p(X) :- X = 1 | c, d.
p(X) :- X = 2 | e, f.
```

The processes `a` to `f` cannot be executed in parallel under our data-parallel execution model. Furthermore, the processor elements cannot be fully utilized—only some of the processor elements execute each clause. A compiler utilizing a join algorithm is effective only when two or more clauses have the same procedure in the body. We can get around this problem if we build an interpreter which executes the instructions of `a`, `c` and `e` simultaneously, effectively emulating a MIMD machine using a SIMD machine. We shall have to store a copy of the program in each processor and interpretation introduces significant overhead, even for programs which do not have this problem. Refer to [NT88b] for an and-parallel concurrent logic programming system taking this approach.

9.1.2 Limitations of the Implementation Scheme

Firebird successfully avoided all inter-processor communication with a processor allocation strategy which allocates equal number of processor elements to each possible alternative. However, this heuristic is much a matter of guesswork. The exact number of processor elements needed by each alternative cannot be found except by actual execution. Parallelism is limited because of the inaccuracy of this guessing especially when the number of processor elements is large. On the other hand, a failed partition will stay idle until backtracking occurs. This again limits the degree of parallelism. Another factor limiting performance is the inherent limitation of or-parallelism—the initial parallelism is small when compared to and-parallelism. For instance, no parallelism can be exploited while the constraints are being generated by the n-queens program. A small degree of parallelism is possible after the first nondeterministic derivation, and it takes some time to reach the maximum degree of parallelism. We have shown the effect of these factors by execution traces in Section 7.9 (page 84).

9.2 Future Work

9.2.1 Extending Firebird

In addition to the indeterministic and nondeterministic derivation steps, we could incorporate a *propagation step* [LPW92] into Firebird. Consider the following example.

```
p(X) :- X >= 3, X =< 4 | true.
p(X) :- X >= 2, X =< 3 | true.
```

Computation fail unless `X >= 2` and `X =< 4`. Therefore, `X >= 2, X =< 4` is told to the store.

Definition 9.1 *Let $G_1, \ldots, G_n$ be the ask constraints of each clause of a predicate. In a* propagation step*, a constraint C, which satisfies the following condition, is told to the store.*

$$D \models \bigvee_{i=1}^{n} G_i \rightarrow C$$

Intuitively, the disjunction of ask constraints of each (satisfiable) clause, or some sound approximation thereof, is told to the store. This extension generalizes the following.

ALPS According to the commit law of ALPS [Mah87], a clause can be committed to if and only if it is validated or it is the only satisfiable clause. Firebird covers only the first part of the commit law—it will only commit to a validated clause. However, if there is only one satisfiable clause, the propagation rule may tell a constraint C which is the disjunction of the ask constraints of the only satisfiable clause, and as a result the clause is immediately validated and committed to.

Domain Independent Propagation The extension provides a convenient syntactic construct for user-controlled *generalized constraint propagation* [LPW92].

Disjunctive constraints in cc(FD) Users may define disjunctive constraints [VHSD93] using a set of clauses with different ask constraints and a `true` body. For example, the following predicate quoted from the *perfect square problem* in [VHSD93]

```
nooverlap(X1,Y1,S1,X2,Y2,S2) :-
    X1 + S1 <= X2 \/
    X2 + S2 <= X1 \/
    Y1 + S1 <= Y2 \/
    Y2 + S2 <= Y1.
```

can be expressed as

```
nooverlap(X1,Y1,S1,X2,Y2,S2) :- X1 + S1 <= X2 | true.
nooverlap(X1,Y1,S1,X2,Y2,S2) :- X2 + S2 <= X1 | true.
nooverlap(X1,Y1,S1,X2,Y2,S2) :- Y1 + S1 <= Y2 | true.
nooverlap(X1,Y1,S1,X2,Y2,S2) :- Y2 + S2 <= Y1 | true.
```

LAIR Users may define looking ahead constraints [VH89] using head matching and ask constraints. For example, in Firebird extended with the propagation step, we may define the logical or function as

```
or(0,X,X).
or(X,0,X).
or(1,1,1).
```

Suppose $X \in 0,1$, $Y \in 0,1$, $Z = 0$, we can deduce $X = 0$, $Y = 0$ using the propagation rule. However, the propagation rule cannot subsume LAIR. This is because Firebird is a flat language and therefore only flat guards can be used for disjunction.

Suppose there are m constraints, $c_1, \ldots, c_m$. $x_1, \ldots, x_n$ are the variables appearing in $c_1, \ldots, c_m$. Let D_i^j be the reduced domain of variable x_i if only c_j is added to the store ($\delta_j = \delta \wedge c_j$). If the disjunction of the m constraints is

told to the store, the following definition by [VHSD93] is the most reasonable approximation of consistency.

Definition 9.2 *The* generalization *of a set of constraint stores* $\delta_1, \ldots, \delta_m$ *is the constraint store*

$$x_1 \in \bigcup_{j=1}^{m} D_1^j \wedge \cdots \wedge x_n \in \bigcup_{j=1}^{m} D_n^j$$

We must repeatedly perform the disjunction until

1. one of the constraints is entailed by the store, resulting in commitment, or
2. all the constraints have failed.

Theorem 9.1 *If a variable* x_i *is missing from one of the satisfiable constraints* c_k*, then the domain of* x_i *cannot be reduced after the disjunction.*

Proof If x_i is missing from c_k, then the original domain D_i will be

$$D_i = D_i^k \subset \bigcup_{j=1}^{m} D_i^j$$

Since the original domain is a subset of the new domain, it cannot possibly be reduced.

Therefore, we make the following observations.

1. If a variable appear in only a subset of the satisfiable disjuncts, we can save execution time by not attempting any pruning of invalid values from that variable.
2. We should discard a disjunct as soon as it becomes invalidated. This makes the optimization in point 1 impossible because in general we cannot detect invalidation without reducing the domain of all the variables in a constraint.

Point 2 is not stated in [VHSD93], possibly because of efficiency reasons.

9.2.2 Improvements Specific to DECmpp

On our implementation platform, each processor in the back-end is much slower than the sequential host. Performance will be improved if sequential parts of the execution can be separated out and performed at the front-end. We propose to optimize the execution before the first nondeterministic derivation step. At the implementation level, we may move low-level operations such as dereferencing and unification to the front-end. An experiment showed that the dereferencing operation was up to 7 times faster when moved to the front-end because local indirect addressing could be avoided and only the memory of a single processor element was accessed[1].

To attain higher performance, the compiler may generate two versions for each predicate, one sequential and the other parallel. The former is used before the first nondeterministic derivation step. At the first nondeterministic derivation step, the heap of the front-end is copied to the processor elements using a garbage collection algorithm. To reduce the overhead of copying, domain variables are always created on the processor elements. In this way, no copying is necessary for programs in which all constraints are generated before any of them are solved.

In addition to these two approaches, we are designing compiler directives to control the execution. For example, the `sequential` directive suppresses the generation of parallel code for a procedure. However, only the code size is made

[1]This is because on DECmpp, processor elements are grouped into clusters of 16 processor elements which share a single memory port. There is a *load solitary* instruction which is optimized for the case when at most one processor element is active in each cluster.

smaller but the execution speed is not improved.

9.2.3 Labeling

Since the size of the domains may be different in each partition, traditional heuristics such as the first-fail principle no longer works. Currently, our implementation is based on generalized forward checking. We are looking for new control strategies which do not depend on the domain size to make decisions. Apart from the heuristic discussed in section 4.4.1, we can choose the most recently changed domain variable to set up a choice point, *etc.* A performance study is needed to be carried out to evaluate such possibilities.

Firebird offers only rudimentary user control over the labeling operation. The user cannot control any other thing than whether a domain variable should be labeled. We did not followed CHIP [DVHS+88] because the introduction of the labeling predicate `indomain/1` into a concurrent language without any top-down, left-to-right execution order will not give the user any more control over the labeling operation than the nondeterministic derivation of the Firebird computation model. The `indomain/1` predicates will not follow any top-down, left-to-right execution order either. It is irrelevant to order the `indomain/1` predicates in a program. [GY92] does not have this problem because in Andorra, the top-down, left-to-right execution order is preserved for nondeterminate goals. In any case, the best solution is a mechanism for user-defined deadlock handlers, like that of Pandora [Bag91]. System predicates giving information on domain size, maximum, minimum and range can be provided to be called by the user-defined deadlock handler.

On the other hand, we may use a plain forward checking strategy, which is

automatic and does not require any user intervention. The system avoids any suspension by labeling the variable which would cause a constraint or goal to suspend on the fly.

Modifying the labeling operation itself is also a possible direction of future work. For instance, we may just divide the domain of the labeled variable into two or more equal parts. We may also divide the domain into continuous chunks (*e.g.* If the domain is {1,2,3,7,8,9,10,11} we may divide it into {1,2,3}, {7,8,9} and {10,11}). In the finite domain part of CLP(BNR) [BO94], a disequality constraint causes a continuous interval to be split into two intervals. It is expected that this can be implemented on a data-parallel machine as follows. Every disequality constraint causes a labeling operation so that the two resulting intervals can be processed in parallel.

9.2.4 Parallel Domain Consistency

By reasoning on variation intervals the minimum and the maximum possible values of a domain variable can be found. Values outside this range are eliminated and this is called *interval consistency*. However, it is not guaranteed to rule out all invalid values. Using the domain consistency technique, each combination of possible values of the domain variables is attempted. This is more effective in eliminating invalid values but consumes more execution time. Unlike cc(FD) [VHSD93] in which the programmer has to specify whether interval consistency or domain consistency is to be used, a Firebird implementation may choose between the two (or some intermediaries) depending on processor elements availability. If there are enough processor elements, each partition will check domain consistency in parallel, resulting in another kind of data-parallelism which can

or in parallel.

Bibliography

[AB91] A. Aggoun and N. Beldiceanu. Overview of the CHIP compiler system. In Koichi Furukawa, editor, *Logic Programming: Proceedings of the Eighth International Conference*, pages 775–789, Paris, France, 1991. The MIT Press.

[ABB92] H. Arro, J. Barklund, and J. Bevemyr. Parallel bounded quantifiers—preliminary results. Presented at JICSLP '92 Workshop on Distributed and Parallel Implementation of Logic Programming Systems, Washington D. C., 1992.

[AK91] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991.

[Bag91] R. Baghat. *Pandora: Non-Deterministic Parallel Logic Programming.* PhD thesis, Imperial College, London, 1991.

[Bla90] T. Blank. The Maspar MP-1 architecture. In *Proceedings of the IEEE COMPCON Spring 1990*, pages 20–24, San Francisco, February 1990. IEEE.

[BM92] J. Barklund and H. Millroth. Providing iteration and concurrency in logic programs through bounded quantifications. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 817–824, ICOT, Japan, 1992.

[BO94] F. Benhamou and W. J. Older. Applying interval arithmetic to real, integer and boolean constraints. *The Journal of Logic Programming*, 1994. To appear.

[BP92] A. K. Bansal and J. L. Potter. An associative model to minimize matching and backtracking overhead in logic programs with large knowledge bases. *Engineering Applications of Artificial Intelligence*, 5(3):247–262, 1992.

[Bra90] I. Bratko. *Prolog Programming for Artificial Intelligence, 2nd Edition.* Addison-Wesley, 1990.

[CCD94] B. Carlson, M. Carlsson, and D. Diaz. Entailment of finite domain constraints. In *Logic Programming: Proceedings of the Eleventh International Conference*, S. Margherita Ligure, Italy, 1994. MIT Press.

[CG85] K. L. Clark and S. Gregory. Notes on the implementation of Parlog. *The Journal of Logic Programming*, 2(1):17–42, April 1985,.

[CG86] K. L. Clark and S. Gregory. Parlog: Parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, January 1986.

[Chr90] P. Christy. Software to support massively parallel computing on the Maspar MP-1. In *Proceedings of the IEEE COMPCON Spring 1990*, pages 29–33, San Francisco, February 1990. IEEE.

[Cla78] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[Coh90] J. Cohen. Constraint logic programming languages. *Communications of the ACM*, 33(7):52–68, July 1990.

[Cra90a] J. Crammond. The abstract machine and implementation of Parallel Parlog. Technical report, Department of Computing, Imperial College, London, July 1990.

[Cra90b] Jim Crammond. Scheduling and variable assignment in the parallel PARLOG implementation. In Saumya Debray and Manuel Hermenegildo, editors, *Logic Programming: Proceedings of the 1990 North American Conference*, pages 642–657, Austin, 1990. ALP, MIT Press.

[CWY91] Vítor Santos Costa, David H. D. Warren, and Rong Yang. The Andorra-I engine: A parallel implementation of the basic andorra model. In Koichi Furukawa, editor, *Logic Programming: Proceedings of the Eighth International Conference*, pages 825–839, Paris, France, 1991. The MIT Press.

[DC93] Daniel Diaz and Philippe Codognet. A minimal extension of the WAM for clp(FD). In David S. Warren, editor, *Logic Programming:*

[KS89] Y. Kanada and M. Sugaya. A vectorization technique for prolog without explosion. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 151–156, 1989.

[KS90] S. Kliger and E. Shapiro. From decision trees to decision graphs. In Saumya Debray and Manuel Hermenegildo, editors, *Logic Programming: Proceedings of the 1990 North American Conference*, pages 97–116, Austin, 1990. ALP, MIT Press.

[KT91] M. Korsloot and E. Tick. Compilation techniques for nondeterminate flat concurrent logic programming languages. In Koichi Furukawa, editor, *Logic Programming: Proceedings of the Eighth International Conference*, pages 457–471, Paris, France, 1991. The MIT Press.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming, Second, Extended Edition.* Springer-Verlag, 1987.

[LPW92] Thierry Le Provost and Mark Wallace. Domain independent propagation. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 1004–1011, ICOT, Japan, 1992.

[LTC93] H. F. Leung, B. M. Tong, and K. L. Clark. The Firebird computation model for finite domain constraint solving in concurrent logic programming and its realization in FD-Parlog. Unpublished manuscript, 1993.

[LvE92] J. H. M. Lee and M. van Emden. Adapting CLP to floating-point arithmetic. In *Proceedings of the International Conference*

on Fifth Generation Computer Systems, pages 996–1003, ICOT, Japan, 1992.

[Mah87] M. J. Maher. Logic semantics for a class of committed-choice programs. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 858–876, Melbourne, 1987. The MIT Press.

[NT88a] M. Nilsson and H. Tanaka. A flat GHC implementation for supercomputers. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 1337–1350, Seattle, 1988. The MIT Press.

[NT88b] M. Nilsson and H. Tanaka. Massively parallel implementation of flat GHC on the Connection Machine. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 1031–1040, Japan, 1988. ICOT.

[Sar88] V. A. Saraswat. A somewhat logical formulation of CLP synchronisation primitives. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 1298–1314, Seattle, 1988. The MIT Press.

[Smi93] Donald A. Smith. MultiLog: Data or-parallel logic programming. In David S. Warren, editor, *Logic Programming: Proceedings of the Tenth International Conference*, pages 314–331, Budapest, 1993. The MIT Press.

[SR90] V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proceedings of the 17th Symposium on Principles of Programming Languages*, pages 232–244, San Fransisco, 1990.

[Tar92a] P. Tarau. Low-level issues in implementing a high-performance continuation passing Prolog engine. Technical Report 92-02, Dept. d'Informatique, Univ. de Moncton, 1992.

[Tar92b] P. Tarau. WAM-optimizations in BinProlog: towards a realistic continuation passing Prolog engine. Technical Report 92-03, Dept. d'Informatique, Univ. de Moncton, 1992.

[TL93] Bo-Ming Tong and Ho-Fung Leung. Concurrent constraint logic programming on massively parallel SIMD computers. In Dale Miller, editor, *Logic Programming: Proceedings of the 1993 International Symposium*, pages 388–402, Vancouver, 1993. The MIT Press.

[Ued85] K. Ueda. Guarded horn clauses. In E. Wada, editor, *Logic Programming '85 — Proceedings of the 4th Conference*, Lecture Notes in Computer Science 221, pages 168–179, Tokyo, July 1985. Springer-Verlag.

[VH89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[VHD87] P. Van Hentenryck and M. Dincbas. Forward checking in logic programming. In *Proceedings of the Fourth International Conference*

on Logic Programming, pages 229–256, Melbourne, 1987. The MIT Press.

[VHSD93] P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation and evaluation of the constraint language cc(FD). Technical Report CS-93-02, Department of Computer Science, Brown University, Providence, 1993.

[War90] D. H. D. Warren. The extended Andorra model with implicit control. Presented at ICLP '90 Workshop on Parallel Logic Programming, Jerusalem, 1990.