AN ABSTRACT MACHINE BASED EXECUTION MODEL FOR COMPUTER ARCHITECTURE DESIGN AND EFFICIENT IMPLEMENTATION OF LOGIC PROGRAMS IN PARALLEL

Manuel V. Hermenegildo

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-86-20 August 1986

AN ABSTRACT MACHINE BASED EXECUTION MODEL FOR COMPUTER ARCHITECTURE DESIGN AND EFFICIENT IMPLEMENTATION OF LOGIC PROGRAMS IN PARALLEL

by

MANUEL V. HERMENEGILDO, E.E., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August, 1986

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Professor G. J. Lipovski, for his continuous support and guidance. He brought the subject to my attention and has always made himself available for discussion and advice. I would also like to thank the members of the Committee, Professors J. C. Browne, J. K. Aggarwal, C. K. Leung, and R. M. Jenevein, for their helpful observations, interest and encouragement, and for the time devoted to the reading of this document.

I am also indebted to Richard Warren and Roger Nasr for many hours of interesting discussion and for their friendship. Richard Warren's implementation of the model described in this document was very useful in proving its viability and in providing performance data. I am grateful to all the other members of the MCC Parallel Processing and Artificial Intelligence groups for their encouragement and to Steve Lundstrom (MCC) and the Fulbright Foundation for their support.

Special thanks to Professor David H. D. Warren for many interesting discussions, for his perceptive comments and suggestions, and for his hospitality. He has been a continuous source of encouragement and inspiration for me.

Finally I would like to thank the many friends who have helped me and encouraged me during these years at The University of Texas. I am very specially indebted to Ian Walker, Pat Cericola, and Evan Tick for their patience in wading through so many earlier drafts of this document and, most importantly, for their invaluable friendship.

Manuel V. Hermenegildo

The University of Texas at Austin
August, 1986

PREFACE

"Sorry to interrupt the festivities",
said HAL, *"but we have a problem."*
Arthur C. Clarke; *2001, A Space Odyssey*

A Word for the Non-Initiated

Scientific reports have an unmistakable tendency to be of a very detailed nature and limited scope. This is largely a consequence of the fact that as our knowledge is broadened we only seem to further unveil reality's inherent complexity. It is in order to confront this complexity that scientific research has moved more and more towards *superspecialization*. However, even though superspecialization seems to be here to stay, the activities of the researcher are hard to justify unless they are motivated (secretly or openly) by some higher-level goal. Of course, such higher level goals often appear obvious to most researchers in their respective fields (in the case in hand, those of Computer Engineering and Computer Science). Computer Scientists and Engineers are therefore urged at this point to skip the rest of this preface and jump with the author into the first chapter. This preface is not intended for them. Instead, it will attempt to offer the "uninitiated" reader in the mysteries of computers, declarative languages and parallelism, both a simple introduction to the subjects treated in the rest of this document and, hopefully, some justification as to why it may make sense to explore these subjects at all.

Computers Need to be Faster and Easier to Use

It is perhaps the fact that computers offer promise to one day mimic at least some of the simpler functions of the human mind (itself undoubtedly one of the most

intriguing "mechanisms" with which we are confronted) that has always drawn our attention towards them. However, computer research is, as so many areas of science and engineering, still far from its most idealistic goals, and, in particular, from that of achieving any kind of "intelligent" behavior from an automaton. Research in "Artificial Intelligence" is faced today with a number of limitations. Firstly, we still lack a clear understanding of *how such behavior could be obtained from a machine.* Secondly, we do not know *how to build computers that are fast enough* that they could provide responses according to that behavior in a reasonable amount of time, and which are at the same time *easy enough to use* that the associated programming tasks would represent feasible endeavors.

The first of the problems mentioned above is one of the many subjects of *artificial intelligence* research. Instead, and as the subject of this dissertation, we will be interested in addressing the second of those limitations, i.e. providing computers that are at the same time *more powerful*, and *friendlier* to the user. Fortunately enough, we do not need to resort to any futuristic quest for intelligence to understand the usefulness of such an endeavor: we already need faster, easier to use machines today, not only for the advancement of artificial intelligence, but also in most other current computer application areas.

1.- Making Computers Easier to Use

Let us consider the issue of making machines *easier to use* first[1] . At our (relatively modest) current state of development in human interaction with computers, our main means for instructing them *what* to do is by writing a *program*, i.e. a list of instructions which are to be executed by it. These instructions are expressed in a particular language that the computer can understand: a *programming language.*

Conventional programming languages generally express these instructions as a series of precise "actions" that are to be performed by the computer one after the

[1]We will treat the issue of *computational power* in the next section.

other[2] . This is known as an *imperative* style, a program being a sequence of commands or statements. Programs today come in this format largely as a consequence of the fact that the first computers were no more than the equivalent of one of today's hand-held calculators and that early programs were just *sequences of the basic instructions that a particular machine could directly execute*[3]. Programming languages then emerged as a tool for making it it easier for a human being to express the actions required from the computer. However, it is a fact that computers already existed *in a particular form* before these languages were designed, and this undoubtedly invited a "machine-oriented" style in these designs which still lingers in today's programming languages. These languages are often so apart from the natural way in which humans think and express themselves that programming a computer is frequently a difficult and error-prone task for any sizeable problem.

The question of course is, can we design a computer language which is free

---

[2]For example, suppose that we want to program the computer to simply generate the squares of all positive integers. One way of doing this is by specifying the *actions* that may be involved in obtaining such a list:

1. *Start with the number 0,*
2. *find the square of the number by multiplying it by itself,*
3. *print the square,*
4. *compute the next number by adding 1 to the previous number,*
5. *go to step 2.*

This would be expressed in less verbose terms (in a *conventional* programming language) more or less as follows:

```
        Number = 0;
loop:   Square = Number * Number;
        print( Number );
        Number = Number + 1;
        goto loop:
```

[3]For example, for an actual hand-held calculator a "program" for adding "4" and "3" would be:

```
press 3
press +
press 4
press =
```

from the *imperative* style? If we avoid any machine oriented considerations, the first language to come to mind is, of course, the human natural language: the user's mother tongue. Such a language though presents a number of serious drawbacks. These drawbacks include its verboseness, only made worse by its vagueness and ambiguity if not provided with a suitable context or a great deal of (normally assumed) knowledge. This fact was already realized by mathematicians long before computers came to being and they devised "Logic" as a means of clarifying and/or formalizing the human thought process. Logic lets us express facts and rules about the world in a precise and concise way and draw conclusions from them which can be formally proven to be correct. Thus, Logic would tells us, for example, that the assumptions

Aristotle *makes* **cookies, and**

Plato *is a friend of* **anyone who** *makes* **cookies.**

imply the conclusion

Plato *is a friend of* **Aristotle.**

Symbolic logic is simply a shorthand for expressing conventional Logic: if we agree that **makes(X, cookies)** means *"X makes cookies"*, $\forall$ **X** means *"for all X"*, **X** $\rightarrow$ **Y** means *"if X then Y"*, and **friend(X,Y)** means *"X is a friend of Y"*, then the example above can be expressed in symbolic logic as

makes(Aristotle, cookies)

$\forall$ **X, makes(X, cookies)** $\rightarrow$ **friend(Plato, X)**

and the conclusion as

friend(Plato, Aristotle)

Clearly, one can mechanically translate from Symbolic Logic to natural language by using a "conversion table" for the symbols like the one provided above.

It is this ability of symbolic logic to express knowledge in a way that is very precise and compact, while at the same time close to the natural way in which humans express themselves that led to the concept of *using Logic as a means for*

programming computers. This idea was first proposed in a formal manner by Kowalski [39] not many years ago and has since received wide acceptance as one of the most promising programming paradigms for future computers[4]. Logic provides more concreteness than a natural language, but it is also far less machine oriented than conventional computer languages. The main difference with them is its **"declarative"** nature: in logic, statements express facts, knowledge about the problem to be solved, rather than precise instructions to be followed step by step[5].

---

[4]For example, Logic Programing is the language of choice in the Japanese Fifth Generation Computer Project.

[5]A "declarative" description of the problem proposed previously would be the following. First, let us *define* the positive integers:

- **0** *is a* **positive integer**.
- **X** *is (also) a* **positive integer** *if it is the result of* **adding 1** *to another* **positive integer** *Y*.

Now let us *define* "square":

- *Y is the* **square** *of X if it is the same as the result of* **multiplying** *X by itself.*

Note that these statements provide our *knowledge* about the problem, rather than a sequence of instructions to be followed step by step. This is much closer to the way things are explained to humans. Just for reference, here is the listing of the same program written in Prolog [58] (a practical, though still far from perfect, "logic programming" language):

```
pos_integer(0).
pos_integer(X) IF pos_integer(Y) AND X=Y+1.

square(Y, X) IF Y=X*X.
```

Now we can ask the Prolog system for the squares of all positive integers:

```
pos_integer(X) AND square(Y, X)?
```

and Prolog will try to find them. One of the interesting things that we also can do now is *ask other types of questions*. For example, the answer to

```
square(Y, 4)?
```

is `"Y=16"`. Surprisingly enough, the answer to

```
pos_integer(X) AND square(4, X)?
```

is `"X=2"`: the same program can be used (even though only in limited cases) to find square roots!

Logic is (with *functional languages*) part of the reduced class of **declarative languages**. It is hoped that the advantages that these languages offer over conventional ones will make computer programming an easier and less prone to error job. It was mentioned how this was one of our objectives. In this dissertation *Logic* (and, to some extent, functional languages) *is chosen as a convenient programming paradigm for new, easier to use computers.*

2.- Making Computers Faster

From the early days of computing, the quest for faster machines has been one of the driving forces of computer engineering research. The availability of the computer made it possible to create applications that were unimaginable before its birth. These applications in turn suggested others which were more and more demanding on computer power, thus producing a snowball effect which today seems to put no end on the demands for computer speed and power.

There are many ways in which a particular machine can be enhanced, but there are always physical limits to the speed at which a given machine can operate: the speed of light and the size of the computer dictate the ultimate limitation, the time involved in moving information from one part of the machine to the other. But even before the limits of a given technology are reached, there is also the question of cost-effectiveness: a point is eventually arrived at in which a moderate increase in performance demands an enormous increase in cost. A concept which can offer a cost-effective increase in performance which can go beyond the limits of sequential systems is **parallelism**: the subdivision of a problem into subtasks which can be executed simultaneously by different agents.

Parallelism is not a concept particular to computers: it is a constant in nature and in the organization of human societies. For example, if a company having only one engineer needs to have a project finished by a given date but the engineer estimates that it will be impossible to complete the project in the given time, the obvious solution (short of firing the engineer, of course) is to hire *more* engineers to work on the project simultaneously and cooperatively so that it can be finished in

time[6]. Such a basic everyday idea can also be applied to computers: *if one computer cannot perform a given task in a certain amount of time, a number of computers can be set to work on the task simultaneously.*

Current **parallel computers** consist of a number of *processors* (each of them more or less a complete computer in itself) linked by some kind of interconnection network which makes it possible for information to be interchanged between them (much in the same way as telephone lines -or simply the human speech- are used by the engineers of the example above). Another typical organization is to provide all processors with common access to the information they are all working on (a *Shared Memory*). This is equivalent to having all the engineers (again in the example above) work on the same set of diagrams simultaneously.

If the task in hand can be separated into relatively independent parts, parallelism can be a simple matter. If the parts of the task are more interrelated, efficiently coordinating the actions of the different processors involved will be a more complicated issue, since there will have to be substantial communication between them in order to inform each other of their current results. This is similar to the periodic meetings that engineers working on a project need to have in order to keep the project well coordinated. The overhead involved in this communication is an important factor to take into account: it is clear that *two* engineers will probably not solve the problem in *half* the time because of the time lost interchanging results (or, if they are working on the same set of diagrams, waiting for the other to finish working with the particular sheet needed). These considerations will be of the utmost importance in the design of *parallel* computers.

Despite the problems involved, parallelism offers an enormous potential in high-performance, cost-effective computer design and is already a reality in the form of many commercial products. Parallelism will also be one of the central subjects of this document.

---

[6]The human brain is another notable example of parallelism: it is built out of a multitude of relatively slow elements, but the whole system has an unequaled information storage and processing power.

Executing Logic Programs in Parallel

In consequence with the considerations presented in the previous sections, this dissertation deals with both the ideas of **parallelism** and **logic programming**. It will try to *provide guidelines in the design of computers that can execute programs which are easier to create* (because they are written declaratively, using "Logic"), *and which are fast enough* (because they use parallelism extensively) *to cope with truly demanding applications*. Although we still are far from our ultimate goals, these new computers could represent an important step in our quest for intelligence. In the meantime, they will provide a friendly and powerful tool to help us cope with our current applications and everyday duties. Let me leave you with this idea for now. I need to go and chat with HAL [1], and thank him for so much inspiration.

ABSTRACT

The term "Logic Programming" refers to a variety of computer languages and execution models which are based on the traditional concept of Symbolic Logic. The expressive power of these languages offers promise to be of great assistance in facing the programming challenges of present and future symbolic processing applications in Artificial Intelligence, Knowledge-based systems, and many other areas of computing. The sequential execution speed of logic programs has been greatly improved since the advent of the first interpreters. However, higher inference speeds are still required in order to meet the demands of applications such as those contemplated for next generation computer systems. The execution of logic programs in parallel is currently considered a promising strategy for attaining such inference speeds. Logic Programming in turn appears as a suitable programming paradigm for parallel architectures because of the many opportunities for parallel execution present in the implementation of logic programs.

This dissertation presents an efficient parallel execution model for logic programs. The model is described from the source language level down to an "Abstract Machine" level suitable for direct implementation on existing parallel systems or for the design of special purpose parallel architectures. Few assumptions are made at the source language level and therefore the techniques developed and the general Abstract Machine design are applicable to a variety of logic (and also functional) languages. These techniques offer efficient solutions to several areas of parallel Logic Programming implementation previously considered problematic or a source of considerable overhead, such as the detection and handling of variable binding conflicts in AND-Parallelism, the specification of control and management of the execution tree, the treatment of distributed backtracking, and goal scheduling and memory management issues, etc.

A parallel Abstract Machine design is offered, specifying data areas, operation, and a suitable instruction set. This design is based on extending to a parallel environment the techniques introduced by the Warren Abstract Machine, which have already made very fast and space efficient sequential systems a reality. Therefore, the model herein presented is capable of retaining sequential execution speed similar to that of high performance sequential systems, while extracting additional gains in speed by efficiently implementing parallel execution. These claims are supported by simulations of the Abstract Machine on sample programs.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

1.1 Computers Today and Tomorrow

One of the most exciting and active facets of Computer Science and Computer Engineering research today seems to be the quest for the "Next" generation machine. The fast pace which characterizes advancement in the area may make it difficult to keep track of the rise and decline of computer generations, but the driving force behind this quest still seems to be the same as in the early days of computing: the need for machines that are *more powerful*, *more cost effective*, and *easier to use*.

The circumstances have today, of course, changed in many ways [79, 80]: **technologically**, various areas, including VLSI Technology, Computer Architecture, Software Engineering, and Artificial Intelligence, seem to be continuously on the threshold of new, major advances. A substantial increase in computational power availability and a greater understanding of how to make more effective use of this power is expected from the contributions in these areas. From the *applications* point of view, the greater appearance in the applications spectrum of non-numerical tasks calls for a move from scientific and raw data processing to symbolic computation: expert systems, knowledge bases, and advanced computer aided design (CAD), etc. seem to be the candidate applications of future systems. These trends are also being supported by **social factors**: interaction with the computer is evolving towards a more "human-oriented" environment. This environment can be expected to

eventually comprise a complex combination of natural language understanding and perception (speech and vision) components, these components themselves requiring a great deal of the computer's resources. Perception can be expected to be present in applications ranging from office systems (for human-machine interaction) to computer integrated manufacture (CIM).

1.1.1 The Top-Down Approach to Computer Architecture

Clearly, there is no reason to suppose that the above mentioned increase in the demand (and, hopefully, the availability) for system functions and performance will come to an end in the near (or far) future. The question is, of course, how these demands can be met. An imaginary "plan of attack" for Computer Science and Computer Engineering research, would probably address at least the following issues:

- Which applications, and computing environments (connection to databases, real time systems, communications, etc.) will the computer have to handle.
- Which algorithms and heuristics will be able to solve these tasks efficiently on the available (i.e., sequential or parallel) computer organizations.
- Which programming languages and computational models will make the task of expressing the algorithms above a tractable problem for humans.
- Finally, which architectures will be able to efficiently implement the computational models and run those applications at the required speed and cost.

Development of the first computers seemed to consider many of these issues: the application was more or less determined (numerical problems) and the algorithms had a well known structure (sequences of numerical computations). The first computer architectures were consequently efficient "number crunchers" and the first computer languages very effective mathematical FORmulae TRANslators. It is an irony that because the most successful such design of all time, the *Von-Neumann Computer*, has also turned out to be rather efficient at many other types of tasks, its principles have

been maintained virtually untouched in the face of differing requirements. Therefore, the original "global design" approach has often since been substituted for an *a priori* acceptance of the existence of the computer in a particular form. The actual unsuitability of traditional architectures for many tasks has sometimes been obscured by their striking ability to *emulate* other machines, while the performance requirements have (thus far) been met to some extent by providing this "emulation" with sufficient raw speed.

There is currently a noticeable change in this trend. There is an awareness that *the issues listed at the beginning of this section are very closely related*, and a number of machine organizations are presently being proposed which are more specifically tuned to particular languages, algorithms, and/or applications. *The explicit consideration of those issues, and in the order listed therein, is often referred to as the* **"top down"** *(or "language first") approach to computer architecture design.*

As evidence of this changing trend, some recent commercially successful products already seem to be the result of this *top down* approach. In these systems, the enormous "gap" often encountered between the semantics of the languages being considered and the underlying architectures on which they are executed, believed to be responsible for their sometimes limited performance, is gradually being reduced by reconsidering the architectural design in the light of language requirements. Such is the case of the current LISP machines (such as the Symbolics3600 [73]) and Logic machines (such as Japan's Personal Inference Machine [53]). RISC [67] [65] architectures are also an example of a design at least partly driven by language considerations. Even though these designs are influenced by the *top down* approach, some of them are still fairly conventional designs. However, the *top down*

approach also results in many other cases in a more radical departure from the Von-Neumann model [21]. Some of these systems (for example, Array Processors) already offer impressive performance, although they are very often limited in their applications.

Of course, however attractive the *top down* idea may be, *bottom up* considerations cannot be completely overlooked in any practical design. For example, as a *bottom up* consideration, both the capabilities and limitations of current technologies have to be taken into account. The influence of this consideration is present in many of the above mentioned designs: the RISC concept is not only justified by the particular implementation needs of current high level languages, but also by the current state of the art in VLSI design and compiler technology. Fortunately, the combination of the two (*top down* and *bottom up*) approaches can result in a *synergetic* effect: as seen above, language semantics can inspire new computational models which can result in novel architectures. Conversely, new architectural ideas can suggest new language concepts, or give new life to old ones which may have been previously discarded as difficult to implement efficiently.

To be consistent with the *top down* criteria, we herein make a conscious assumption regarding the first two issues listed therein, namely applications and algorithms. This assumption is that *future computing applications will be predominantly symbolic in nature*. This assumption is based on the considerations presented in the previous section. With this in mind, in the following sections we will introduce some criteria currently supported by many researchers for addressing the other two issues of the *top down* approach, programming environments and architecture design. As mentioned previously, despite changing circumstances, the objective in these areas still seems to be the same as in the early days of computing: the design of machines that are *more powerful*, *more cost effective*, and *easier to use*.

1.1.2 Improving Programming Environments

Computer Languages are our present means of instructing computers to do what we want them to do. The importance of the characteristics of these languages which determine the *ease* and *precision* with which humans can accomplish this job clearly cannot be underestimated. However, other characteristics of these languages often determine the *efficiency* with which the problem can be solved on a particular machine. In the *top down* approach, a high priority is given to the first consideration and computer architecture design is deemed responsible for the solution of the efficiency issue.

1.1.2.1. Procedural vs. Declarative Languages

Most languages presently in use -FORTRAN, PASCAL, C, BASIC ...- are **procedural** in nature, that is:

- Computations are performed in a predetermined order which is explicitly expressed in the program.
- Each statement is only one step in the algorithm.
- The correctness of each statement (except for perhaps its syntax) cannot be determined without reference to the run-time state of a machine.

These characteristics, often also referred to as the *imperative* style, are largely a consequence of the fact that today's languages are simply an evolved version of the programming style of the early days of computing, when programs were only sequences of the elementary instructions that a machine could directly execute. This "step by step" character still pervades Computer Science even today.

The advent of *functional programming* (based on the Lambda Calculus [10]) and its first implementation in the LISP language [46] brought the new concept of **Declarative** or **Non-procedural** languages to the computing arena. Declarative languages aim at *describing* the *structure* of a problem. In these formalisms we aim

at expressing our "knowledge" about the problem rather than providing step by step instructions for the computer to follow. This is done by specifying our knowledge in the form of **functions** (in *Functional Languages*) or as **sentences of first order predicate logic** (in *Logic Languages*). Some of the benefits of these languages were eloquently defended by Backus in his 1978 Turing Award Lecture [3]. In their ideal form they offer many attractive characteristics which contrast with those of their procedural counterparts:

- They can be read as a formal description of a problem. In this way, a program can be its own specification.
- They are comprised of statements whose order is in general not relevant.
- Programs can be developed in a piecemeal fashion: the "divide and conquer" idea of structured programming is taken one step further, to the statement level.
- Furthermore, the statements of the program can be proved to be valid without considering their relation to other statements in the program.
- They are also less prone to errors, because they do not rely on features which only make sense in machine-level terms (such as side-effects).

Although Logic Programming and Functional Programming have in reality much in common, they each have their defendants and detractors. Some arguments frequently used in favor of functional programming are its support for infinite data structures and streams (i.e. the incremental transference of function arguments), and the concept of "higher order functions" (functions which can be passed around as arguments of other functions). Arguments in favor of Logic Programming are its inherent non-determinism, the support of relations, and the power of the "Logical Variable". These concepts will be explained in later chapters. Today, a *growing acceptance of the validity of both formalisms* is starting to arise: their characteristics appear to address different kinds of problems and, therefore, their

coexistence seems to be justified. Moreover, the quest for a truly general symbolic programming language, being pursued simultaneously from both sides of the controversy, now appears to *converge* towards the common goal of a Logical/Functional Language which would *combine* the advantages of both approaches [66, 23, 35]. For the rest of this monograph we will be mainly concerned with Logic Programming, though many of our conclusions will be equally relevant in the domain of Functional Languages and, of course, clearly applicable to the "Logical side" of an eventual "marriage" between Logic and Functions.

1.1.2.2. Logic Programming

As briefly mentioned before, under the name of "Logic Programming" we refer to the family of declarative languages whose statements are sentences of first order predicate logic. It is fairly straightforward to understand how predicate logic can be used to describe knowledge about a particular problem and to infer conclusions from it, since that was its original design goal: to be a formalism for clarifying and/or formalizing the human thought process. This knowledge is expressed in the form of *facts* and *rules* which are true for a particular problem. Valid conclusions can be then inferred from this knowledge. Thus, the intuitive idea behind logic programming is to provide enough information about the problem in the program that the computer will be able to solve it without the programmer having to worry about the details of *how* it actually comes to its conclusions. Let us illustrate this with an example, written in Prolog (read ":-" as *"if"*):

```
father(john,peter).
father(john,mary).
father(peter,mike).

mother(mary,david).

grandfather(X,Y):- father(X,Z), father(Z,Y).
grandfather(X,Y):- father(X,Z), mother(Z,Y).
```

In this example, some *facts* (that John is the father of Peter, that Mary is the mother of David, ...), and two *rules* (defining that X is the grandfather of Y if there is a Z whose father is X and which is the father -or mother- of Y), are given to the system. This, in turn, is able to answer the question "Is John the grandfather of Mike?" by inferring the correct answer from the facts given and the rule:

```
grandfather(john,mike)?
YES.
```

Each of the lines in the example above is called a (Horn) **Clause**[7]. A **Logic Program** is a set of such clauses. Clauses are generally composed of a **Head** (the part before the ":-") and a **Body** (the part after the ":-"). The body in turn is composed of **Goals**. A clause with an empty body is called a **Fact**, otherwise it is called a **Rule**. A set of clauses with the same "name" (for example the two "grandfather" clauses above) is called a **Procedure**.

The closeness of Logic to the structure of the human natural language, coupled with its conciseness and declarative nature, seem to make it an ideal choice as a computer language from the human point of view. In order to make this a reality, however, a simple mechanism had to be devised which would make it possible for a computer to *automatically* come to a conclusion such as that above. The decisive step towards achieving this goal in practice was provided by Robinson [68] with the discovery of the *resolution principle*. This principle provides an *inference rule* which, when applied repeatedly, makes it possible to *automatically* prove that a given conclusion is a valid deduction from a set of given facts and rules. Kowalski's pioneering work [39], gave a *procedural interpretation* (based on the resolution

[7]More formal definitions of these concepts will be given in the following chapters.

principle) to a subset of first order logic: Horn Clauses. The idea of Logic Programming was born, and it was made a reality by Colmenauer et al. in 1972 [69] in the first implementation of the most popular logic programming language today: Prolog (PROgramming in LOGic). The performance of Warren's Prolog interpreter/compiler for the DECsystem-10 [58] finally proved the usefulness of logic as a practical programming tool.

1.1.2.3. Logic and Control

It should be obvious from the above description that there are two distinct elements in the execution of a Logic Program (and, in fact, in that of any other declarative language):

1. The *program*, i.e. the set of rules and facts, provided by the user.
2. An *evaluator* of the program, which is able to derive conclusions which are consistent with the program.

This distinction was succinctly expressed by Kowalski in the following equation [40]:

```
Algorithm = Logic + Control
```

It should be made clear at this point that the *evaluator* of the program has in principle an enormous degree of freedom ("non-determinism") in selecting which deduction paths to follow while solving a problem. For instance, in the family relationship example above, the *evaluator* is free to choose either of the "grandfather" clauses in its attempt to solve the query. Independently of which clause is chosen, if the choice does not lead to a solution, then the other clause will eventually be tried also. This type of non-determinism is referred to as **"don't know" non-determinism**, as opposed to **"don't care" non-determinism**, in which, once a choice is made, the system is *committed* to that choice and the other paths will not be tried.

The policy that the program *evaluator* uses in chosing one or another of these paths is called the *control strategy*. The existence of a *control strategy* component in a declarative programming language is what makes it different from a declarative formalism (like Horn Clause Logic or Lambda Calculus) [30] . Ideally, the program *evaluator* can be imagined as an autonomous unit, able to solve the problem with no additional help from the user. This separation permits database users or novice programmers, for example, to state only the logic part, leaving the control component to the computer, as we did in the example in the previous section. In practice, however, trying to implement this behavior in more complicated cases often results in very inefficient execution. Thus, most Logic Programming Languages offer the programmer some means of *guiding* the *control strategy*. This is done in some languages explicitly through the use of language constructs ([14], [13], [72], etc.). In others it is done implicitly. For example, in Prolog, much of the control information is encoded in the order in which predicates are written.

1.1.3 Improving Computer Power vs. Cost

Having considered the subject of making computer programming an easier task, let us now shift our attention to the issue of performance. Independently of the programming formalism in use, many different types of architectural organizations are being considered today in the design of more powerful computers. Some of them are:

- A single fast processor with memory hierarchy (frequently with extensive pipelining) (e.g. [22]).
- Multiple processors sharing a common memory (e.g. [71]).
- Multiple processors with private memories and communicating via messages through a network (e.g. [78]).
- Data driven computers, where computations are distributed at the basic function level and control is provided by the flow of data through the system (e.g. [2]).

- Graph reduction architectures, where the elementary operations are reductions on a graph representation of the program and data (e.g. [54]).
- Massively parallel machines, which promise to perform computations with the statistical behavior of a large network of highly interconnected nodes (e.g. [21]).
- Reconfigurable architectures, which can theoretically be reorganized dynamically to behave as some of the above, depending on the granularity level (e.g. [44], [70]).

Most of the above mentioned architectural approaches have at least one thing in common: the presence of parallelism. This is often based on the conviction that some of the objectives of today's systems and those of the future, such as performance and fault-tolerance, can only be achieved in a parallel environment, and also on the fact that today's technology is finally ready to tackle the design of *cost-effective* parallel computers. The main problem with the single processor Von-Neumann computer, despite the relative advantage of its conceptual compatibility with existing systems, is that any effort to increase its performance is bound to hit the speed ceiling of current technology and eventually that of light. But even before the limits of current technology are reached, there is also the question of cost-effectiveness: a point is eventually reached in which a moderate increase in performance demands an enormous increase in cost.

Parallelism is a concept which can offer a cost-effective increase in performance without pushing the limits of technology. Initially confined to the internals of otherwise relatively conventional systems (in the form of *pipelining*), it is already being widely used in a much larger scale in special purpose systems (such as Vector/Array processors) which, although limited in application, offer very impressive cost/performance ratios. General purpose multiprocessor products are now starting to

be commercially available (in shared- [71] or distributed memory [78] designs) which also offer very interesting cost/performance for a much wider application spectrum. Today, the best *long term* prospect in computer architecture research seems to be the development of highly parallel *scalable* architectures which could theoretically be configured to provide any required level of performance by simply adding a sufficient number of elements (or "building blocks") from a *fixed* set.

1.1.4 Parallelism, Logic Programming, and Synergy

The potential for cost effective performance improvement present in parallel architectures, and *the realization of the complexity involved in programming such architectures* has spurred new interest in declarative languages and their computational models. A key issue which is responsible for this interest is the potential *separation of logic and control* which these languages offer. As mentioned before, this separation between *what* has to be done and *how and in what order* it is to be done makes it theoretically possible to write programs in such a way that they can afterwards be executed using different control strategies. The fact that the "freedom" which the program *evaluator* thus has in choosing execution paths *often includes the possibility of executing several of these paths in parallel* makes declarative languages particularly attractive for parallel implementation.

In **Functional Programs** the program *evaluator* can basically exploit two kinds of parallelism [36, 30]: the concurrent evaluation of the arguments of a functional expression (*restricted parallelism*), and the concurrent evaluation of a functional expression and one of its arguments (*stream parallelism*). **Logic Programs** seem to offer even more sources of parallelism as a result of their non-determinacy and declarative semantics [19]. The two basic sources now are **AND-Parallelism** (*the parallel execution of goals in the body of a clause*) and

OR-Parallelism (*the parallel execution of several clauses in a procedure*)[8] . Other lower-level types of parallelism have also been identified, such as *Search Parallelism* (the program is divided into disjoint sets of clauses so that the search for a given clause can proceed in parallel over the different sets), and *Unification Parallelism* etc. AND- and OR-Parallelism in turn can be combined in several ways giving rise to a number of different *forms* of parallelism [30]:

- *Pure OR-Parallelism*: the parallel evaluation of several clauses of a procedure.
- *All Solutions AND-Parallelism*: the parallel evaluation of goals, each of them working on a *different* potential solution.
- *Stream AND-Parallelism*: the parallel evaluation of two goals which share a variable, with the value of the variable being incrementally communicated between them.
- *Restricted (Goal Independence) AND-Parallelism*: the parallel evaluation of several goals in the body of a clause, which are at some point determined to be independent.

It is hoped that the independence of the *control* component will make it possible for the program *evaluator* to take advantage of this potential for parallelism while at the same time automatically keeping track of the communication, synchronization and concurrency issues associated with parallel execution. The programmer will thus be relieved from what will likely be an *impossible* task when a *large* number of processors are used cooperatively in solving the same problem. It is also hoped that it will thus be possible to improve performance simply by "adding" processors or "gracefully" degrade it by removing some of those processors (for example because they are needed in a higher priority task or simply because of hardware faults) in a user-transparent way.

---

[8] These concepts will be further explained in the next chapters.

Therefore, there is a dual relationship between logic programming and parallelism, which represents an example of *synergy* between the *bottom up* and *top down* approaches to computer architecture. Following *bottom up* considerations, parallel execution seems to be the current technological solution to the bottlenecks of conventional architectures, but it presents a programming challenge. Declarative languages and, in particular, Logic languages appear as a possible answer to this challenge. Also, from *top down* considerations, Logic Programming seems to have the potential for meeting the programming challenges associated with new applications and algorithms, but it has traditionally suffered from limited performance. Parallel execution appears as the most promising solution in order to provide the required computational speed at a reasonable cost.

1.2 The Dissertation

This document is largely in the spirit of the top down approach, but a clear effort is also made to give due consideration to technological and other *bottom up* limitations. Following *top down* criteria, we assume the essentially symbolic nature of future computing and the suitability of declarative languages (and, in particular, of Logic Programming) as a symbolic programming paradigm. The subject of this dissertation is to address the remaining issues of the *top down* approach: execution models and architectures. It will offer an *efficient parallel execution model for Logic Programs*, specified *down to the abstract machine level.* The research approach chosen in order to achieve this goal will be described in the following section.

1.2.1 Research Approach

Two basic decisions have determined the research approach taken:

- The choice of the type of parallelism being implemented.
- The choice of the type of abstract machine underlying the implementation.

Concerning the first point above, we have seen in previous sections how Logic Programs offer many different sources of parallelism. Ideally, all these sources should be exploited simultaneously in a given system. Nevertheless, the management and control of this parallelism is non-trivial and the overhead involved in exercising these management functions could very well completely overshadow any performance gains obtained through parallel execution. If efficiency is an important issue in the design, due consideration has to be given to the run-time cost associated with the implementation of the different types of parallelism which are chosen to be supported.

Implementation of **OR-Parallelism** is, at least in principle, relatively straightforward since the parallel processes involved are independent. Thus, there are several proposals which include this type of parallelism [11] [55]. However, a copy of the complete state of the computation up to the branching point has to be given to each of the alternative paths being evaluated, and independent binding environments kept for each one of them from then on. This scheme can clearly require excessive amounts of storage and/or copying time although this overhead can be limited with the use of specialized hardware. In a more efficient implementation scheme only parts of the environment which are to be written by alternate clauses need to be copied while other parts can be shared[9] . The main problem in OR-Parallelism, however, is

---

[9]An implementation of OR-Parallelism supporting this scheme and based on the Abstract Machine proposed by Warren [88] has been realized by R. Overbeek et al. [55] at Argonne Labs. Warren has also proposed a model based on *"hash windows"* as reported in [7].

the combinatorial explosion in the size of the process tree generated. This is aggravated by the fact that sometimes, if only one solution is needed, much of the computation cannot be considered "useful work". Solutions proposed for this problem include the use of annotations in order to restrict the generation of OR-parallel alternatives [55] and the use of heuristics in order to prune as many of the paths not leading to a solution as possible early in the computation [41, 45]. OR-Parallelism is useful in programs which are heavily non-deterministic as, for example, in search based applications.

AND-Parallelism, on the other hand, offers potential for performance improvement even in highly deterministic programs. Unfortunately, AND-parallelism presents a series of problems which have for some time limited its application to only trivial cases. Most of these problems arise from the fact that goals in the body of a clause which are candidates for AND-parallel execution often share variables between them and are therefore not independent. A *variable binding conflict* appears if various goals attempt to bind a shared variable to different values.

One solution to this problem is to determine one goal as the *producer* of the variable, and the others as consumers. In stream AND-Parallelism these goals all run in parallel and the value of the variable is *incrementally* passed ("pipelined") from the producer to the consumers. This is useful in that it allows the description of systems of communicating processes. However, the low level of granularity involved in stream parallelism seems to make it difficult to implement in an efficient way. The main drawback in stream AND-Parallelism, however, is that it is very difficult to implement in the presence of non-determinism. Therefore, recently proposed systems which exploit this type of parallelism *give up true non-deterministic search* by implementing "committed-choice" (i.e. "don't care") non-determinism. Such is the

case in PARLOG [30], Concurrent Prolog [72], and Guarded Horn Clauses (GHC) [81]. Once a path in the execution tree is chosen, no other paths will be explored. These systems are somewhat closer to functional languages in the sense that clauses behave as functions, providing only *one* solution path to a given query.

While committed choice non-determinism has solved a number of practical implementation problems, "don't know" nondeterminism is generally regarded as one of the most interesting features of Logic Programming. Most "committed choice" systems recognize this fact and some (e.g. PARLOG) include the possibility of invoking a "don't know" non-deterministic subsystem within the language, though this will in general default to sequential execution. Alternative approaches for overcoming the lack of "don't know" non-determinism using program transformations generated through partial evaluation have been proposed by Codish [18] and Ueda [82]. However, there is an alternative way of dealing with variable binding conflicts which naturally supports both AND-Parallelism and "don't know" non determinism: restricting AND-parallel execution to sets of goals which are *determined to be independent at run-time.*

Although detecting and dealing with variable binding conflicts has previously needed extensive user-annotation and/or excessive run-time overhead [20], we will show in this dissertation that AND-parallelism supporting full non-determinism can in fact be implemented very efficiently. Thus, we will be mainly concerned with the implementation of AND-Parallelism in the presence of "don't know" non-determinism. The high overhead previously associated with the determination of *goal independence* will be greatly reduced in this model by combining a generalized version of Restricted AND-Parallelism [25], *Goal Independence Parallelism* [32], with some of the implementation techniques of current high performance sequential systems.

This brings us to the second decision regarding the research approach taken: the choice of the type of abstract machine underlying the implementation. Regarding this point, an *evolutionary* approach is chosen: the execution speed of sequential logic programming systems has been constantly improving since the appearance of the Marseille implementation. Warren's Prolog interpreter/compiler for the DECsystem-10 [58] proved that logic programming could offer performance levels comparable to those of functional languages on conventional architectures. Today Prolog runs on a desktop personal workstation at speeds comparable to those of the DECsystem-10 implementation [64] [76], and pipelined architectures [77] and microprogrammed Prolog machines [26] offer promise to approach the 1Mlips (Logic Inferences per Second) line. Most of these implementations are based on the Abstract Machine proposed by Warren [88] (the "**WAM**") which has made very fast and space efficient systems possible.

In spite of the great advances achieved by *sequential* systems, further improvements are necessary in order to meet the requirements of present and future applications. As we have seen in the previous sections, the source for this performance improvement is parallel execution. However, logic programs, in addition to offering considerable opportunities for parallelism, often also present code segments requiring sequential execution. A system which can support parallelism while still making use of the optimizations offered by current systems (at least during sequential execution) is thus highly desirable. This is the approach taken in our design: *to provide the mechanisms for supporting forward and backward parallel execution of logic programs as extensions to the ones used in a high performance Prolog implementation.*

The main advantages of this approach then are: first, sequential execution is

still as fast and space efficient as in the high performance Prolog implementation (modulo some minimal run-time checks); second, because the model is offered in the form of *extensions*, which are fairly independent, in spirit, from the peculiarities of that implementation, the techniques which will be developed will be applicable to a variety of compilation/stack based models. For example, they could be applied to the "don't know" non-determinate subsystem of a committed choice model. Finally, the conceptual similarity with traditional code makes it possible to make use of existing compiler technology.

1.2.2 Purpose of the Dissertation

In brief, the purpose of the dissertation is to design an *efficient* parallel execution model for Logic Programming implementation and computer architecture design. The criteria to be met by this execution model include the following:

- It should support AND-parallel execution and "don't know" non-determinism simultaneously.
- Variable conflicts should be detected and treated with a minimum of run-time overhead.
- It should support all the optimizations offered by high performance sequential implementations for sequential code segments, and as many of them as possible during parallel execution.
- The techniques involved should be applicable to other similar models.
- Control should be completely distributed. Treatment of issues such as scheduling and memory management should be taken into account.
- Also, control issues should be transparent to the user, so that the same program can be executed on any number of processors with the only noticeable difference being a variation in performance.
- The model should be precisely specified, at least to the abstract machine level, so that it can be implemented or realistically evaluated through simulation.

- It should prove efficient in processor and memory resources.

1.2.3 Contributions

The result of the research is the execution model proposed in the previous section. This execution model is described from the source language level down to the abstract machine level. An evaluation of its performance, and implementation and architectural design considerations are also presented. The main original contributions are:

- To prove that Goal Independence AND-Parallelism can be efficiently implemented in the presence of "don't know" non-determinism.

- To present a formal description of a new set of Conditional Graph Expressions (**CGE**'s) which control this type of parallelism and a suitable embedded syntax for them.

- To provide precise forward and backward procedural semantics for Logic Programs annotated with **CGE**'s which can be implemented efficiently. This includes a distributed, "semi-intelligent" form of backtracking.

- To develop a distributed stack execution model based on the above semantics, show the goal scheduling and memory management issues associated with such a model, and offer solutions to them.

- To provide an abstract machine level implementation of the model which offers similar optimizations to those of high performance sequential systems. The abstract machine is presented in the form of *extensions* to one of the highest performance current sequential implementations, the Warren Abstract Machine (**WAM**). New mechanisms (the *parcall frame* and the concept of *markers*) are introduced for controlling AND-parallel execution and distributed backtracking, and for the division of stack sections.

- To present a complete instruction set at the abstract machine level and indications showing how to compile Logic Programs into this instruction set.

- To show the efficiency of the Abstract Machine through simulations.

1.2.4 Dissertation Outline

This dissertation can be viewed as comprising two parts:

Part 1 (chapters 2 through 5) is an introduction to parallelism in Logic Programming and a general description of the parallel execution model in which the abstract machine design is based:

- Chapter 2 is a brief introduction to basic notions of Computational Logic leading to the idea of programming in Logic. Prolog is presented as an example of a Logic Programming Language.

- Chapter 3 deals with the relationship between Logic Programming and parallelism. The sources of parallelism present in Logic Programs, the problems associated with their implementation, and some previously proposed solutions for these problems are introduced. Finally, *Goal Independence* is chosen as the primary source of parallelism in this implementation.

- Chapter 4 starts the actual description of the execution model. Forward and backward procedural semantics are offered for a very general model of goal independence: Horn Clauses annotated with Conditional Graph Expressions. Strategies for checking and generating these expressions and a programmer's view of the system are also presented.

- Chapter 5 first presents a simple, distributed stack memory management model, a goal scheduling strategy, and a processor state diagram, and studies the interactions between these elements. Possible implementation problems are then studied and alternative algorithms are proposed for different cases of granularity and processing element number and complexity.

Part 2 (chapters 6 and 7) is of a more detailed nature and it deals with the actual implementation of the execution model at the abstract machine level:

- Chapter 6 introduces some of the basic techniques used to support the algorithms described in chapters 4 and 5 at the abstract machine level. These techniques are shown to be compatible with those of current high performance implementations. Two basic models of backtracking after local execution of of sibling goals, the "LGF" and "Marker" models, are introduced.

- Chapter 7 describes the data areas and instruction set of an Abstract Machine as extensions to the **WAM**. This abstract machine is shown to support parallel execution of Logic Programs based on the algorithms presented in previous chapters. The "Marker" model described in Chapter 6 is used in this design.

Chapter 8 finally offers conclusions and suggestions for future work. Related material to the rest of the chapters is presented in the appendices:

- Appendix A offers a set of examples of compiled code for the Abstract Machine of Chapter 7, explaining how to deal with several cases such as complicated conditions in the **CGE**'s and nested **CGE**'s.

- Appendix B lists some of the results obtained from the simulations which were used to evaluate the performance of the Abstract Machine. It also provides details about the simulator itself and the test programs used.

Chapter 2

Logic Programming

This chapter is a brief review of basic notions of Computational Logic leading to the idea of programming in Logic. The intention of this chapter is simply to place the subjects which will be described in the rest of the dissertation in perspective with respect to well established concepts in Logic Programming, but it can also be considered a short tutorial in these subjects. For a more extensive introduction to general aspects of Computational Logic, the reader is referred to Kowalski's classic, *Logic for Problem Solving* [40]. The book by Hogger [34] offers an excellent introduction to many aspects of Logic Programming from theoretical issues to implementation techniques. The books by Clocksin and Mellish [16] and Clark and McCabe [15] are recommended as tutorials on Prolog programming.

2.1 Logic

Logic[10] provides a formal way of representing assumptions and conclusions about any domain, and of dealing with the relationship of implication between them. Symbolic logic is essentially a shorthand representation for traditional logic. Its basic elements are **constants**, **variables**, **functors**, and **predicate symbols**. *Constants* represent fixed objects (such as "table", "john", "3" ...) while *variables* stand for arbitrary objects (i.e. "X"). A **term** is recursively defined as a constant, or a variable, or an expression of the form

[10]Throughout this chapter the term *Logic* refers to "First order predicate logic".

$$f(t_1, \ldots, t_m)$$

where f is a *functor*, $t_1, \ldots, t_m$ are terms, and m, the "arity" of the functor, is $\geq$ 1. In this way functors are used to create **compound terms** (also called *structured terms*). An **atomic predicate** is an expression of the form

$$p(t_1, \ldots, t_k)$$

where p is a *predicate symbol*, $t_1, \ldots, t_m$ are terms, and k, the "arity" of the predicate symbol, is $\geq$ 1. Atomic predicates represent *relationships* between terms. *Constants, variables, functors,* and *predicate symbols* can be any mutually disjoint sets. In order to distinguish a constant from a variable, we will give variables names starting with a capital letter[11]. Other elements can be identified from their relative positions.

Atomic predicates, in turn, can be combined by using **logical connectives** ($\wedge$ ("and"), $\vee$ ("or"), $\rightarrow$ ("implication"), $\neg$ ("negation") ...). In addition, the scope of the variables in these predicates can be delimited through **quantifiers** ($\forall$ ("universal"), $\exists$ ("existential")). The resulting **compound predicates** are the basic sentences of Symbolic Logic. Figure 2-1 shows some examples which should illustrate these definitions.

2.1.1 Clausal Form

The complete set of elements described above (all logical connectives + all quantifiers) is redundant: some of them can be expressed as combinations of the others. Using this property compound predicates can be conveniently expressed in the simplest possible forms. One of the most interesting is **Clausal Form** where each compound predicate is expressed as a set of simple **clauses**, each of which has the form:

---

[11] In order to preserve compatibility with conventional Prolog syntax.

Terms:

mary	[constant]
X	[variable]

Functors:

employee

Compound Terms:

employee(plato, salary(1500, gold_coins), position(philosopher))
(The arity of "employee" is 3)

Predicate Symbols:

father_of

Atomic Predicates:

father_of(peter, mary) ("Peter is the father of Mary")

Logical Connectives:

$\neg$	[negation]
$\wedge$	[and]
$\vee$	[or]
$\rightarrow$	[implication]
$\equiv$	[equivalence]

Quantifiers:

$\forall$	[universal]
$\exists$	[existential]

Compound Predicates:

$\forall$ X, $\exists$ Y program(X) $\wedge$ procedural(X) $\rightarrow$ hasbug(X,Y)
("All procedural programs have at least one bug")

Figure 2-1: The Elements of Logic

$conc_1, \ldots, conc_m \leftarrow cond_1, \ldots, cond_n$

where $conc_1, \ldots, conc_m, cond_1, \ldots, cond_n$ are atomic predicates, and n, m are ≥ 0. The atomic predicates $conc_1, \ldots, conc_n$ are called the **conclusions** of the clause, and each comma separating them represents an "$\vee$" ("or") connective. The atomic predicates $cond_1, \ldots, cond_n$ are called the **conditions** of the clause,

and each comma separating them represents an "$\wedge$" ("and") connective. All the variables in all the predicates are implicitly quantified by "$\forall$" ("for all"). Thus, if the clause contains the variables X_1, ..., X_k clausal form is really simply a shorthand for

$$\forall X_1, \ldots, \forall X_k \; conc_1 \vee \ldots \vee conc_m \leftarrow cond_1 \wedge \ldots \wedge cond_n$$

and it can be read as

for all X_1, ..., X_k,
$conc_1$ *or ... or* $conc_m$
if $cond_1$ *and ... and* $cond_n$

If n=0 then it can be read as

for all X_1, ..., X_k,
$conc_1$ *or ... or* $conc_m$ *are always true.*

and if m=0 then read

for all X_1, ..., X_k,
$cond_1$ *and ... and* $cond_n$ *are always false.*

Also, if m=n=0 then the clause represents the predicate that is always false, and it is written as $\perp$.

Clausal form has the advantage over the standard form of Logic of being simpler and more concise, while still allowing the representation of all predicates which can be expressed in standard form. The process of converting from standard form to clausal form is straightforward and well known. A (Prolog) program which performs this task automatically is shown in [16][12].

---

[12]For example, if we apply such an algorithm to the following example,

$\forall$ X, $\exists$ Y, program(X) $\wedge$ procedural(X) $\rightarrow$ hasbug(X,Y)

we would obtain:

hasbug(X, bug(X)) $\leftarrow$ program(X), procedural(X)

2.1.2 Resolution

Logic also provides mechanisms for deriving valid conclusions from a set of axioms in a step by step manner. Each of these steps is called an **inference step**, the ordered list of all those steps is called a **proof**, and the mechanisms used for deriving each step are called **inference rules**. Although Logic provides a variety of inference rules, there is one rule which, when applied repeatedly, and without the need to make use of any other rule, can prove that a given conclusion follows from a set of assumptions, provided both the conclusion and the assumptions are all written in clausal form. This rule is **resolution** [68].

One of the basic mechanisms used by resolution is **unification**. Two atomic predicates $p_a(ta_1, \ldots, ta_m)$ and $p_b(tb_1, \ldots, tb_n)$ are said to be *unifiable*, if they have identical predicate symbols (i.e. $p_a \equiv p_b$), they have the same arity (i.e. n=m), and all their terms are pairwise (i.e. ta_1 vs. tb_1, ta_2 vs. tb_2 etc.) *unifiable*. Two terms, ta and tb are unifiable if the following recursive algorithm succeeds for them:

1. **if** ta *is a variable which appears in* tb FAIL[13]; **else**

2. **if** ta *is a variable, and* tb *is not, then* SUCCEED, *and* substitute tb *for all occurrences of* ta; **else**

3. **if** *both* ta *and* tb *are variables, then* SUCCEED, *keeping them as variables, but giving them the same name. These variables are said to share: if a substitution is done for one of them it will also be done for the other;* **else**

4. **if** ta *is a constant then,* **if** tb *is a constant and both constants are identical,* SUCCEED, **else** FAIL; **else**

---

[13]This "check" (referred to as the *occurs check*) is sometimes omitted in practical implementations because of the overhead involved in performing it.

5. *then* `ta` *is a structure (compound term); then,* **if** `tb` *is also a structure, they have identical functors and arity, and all their respective terms are unifiable (using this algorithm recursively),* SUCCEED*;* **else** FAIL.

```
father(john,peter) ←
father(john,mary) ←
father(peter,mike) ←

mother(mary,david) ←

grandfather(L,M) ← father(L,N), father(N,M)
grandfather(X,Y) ← father(X,Z), mother(Z,Y)
```

Figure 2-2: A Family Relationship

Let us illustrate this with an example. Recall the "set of axioms" (expressed in clause form) regarding family relationships presented in Chapter 1 (and reproduced in figure 2-2). In this example, in the clauses

```
father(john,mary) ←
grandfather(X,Y) ← father(X,Z), mother(Z,Y)
```

the atomic predicates `father(john,mary)` and `father(X,Z)` are *unifiable*: they have the same predicate symbol (father) and arity (2), and their terms are unifiable, using the second rule of the *unification algorithm*, with the substitutions (read "/" as "is substituted by") `X/john` and `Z/mary`.

If we have two clauses, such that one of the predicate symbols to the right of the " ← " in one of the clauses is the same (and with the same arity) as one of the predicate symbols to the left of the " ← " in the other clause, we define these two predicates as *complementary predicates*. If these two predicates are also unifiable (as `father(john,mary)` and `father(X,Z)` in the example above) we call them *unifiable complementary predicates*.

We now have all the tools needed for resolution. What resolution basically tells us is that, *given two clauses*[14] *if we build a new clause by listing to the left of the " ← " all the predicates to the left of the " ← " in* **both** *of the original clauses, and doing correspondingly with those to the right, the clause that we obtain logically follows from the two original clauses.* Thus, from

```
father(john,peter) ←
mother(mary,david) ←
```

we can infer

```
father(john,peter), mother(mary,david) ←
```

However, if the two clauses have *unifiable complementary predicates*, the resulting clause is built the same as before, but *leaving out the complementary predicates in both sides and propagating the substitutions made by unification to the rest of the resulting clause.* Thus, from

```
father(john,mary) ←
grandfather(X,Y) ← father(X,Z), mother(Z,Y)
```

we can infer

```
grandfather(john,Y) ← mother(mary,Y)
```

which logically follows from the original two clauses.

Each such application of resolution is a **resolution step**. Resolution is a *correct* inference rule: repeated application of resolution will always give us valid clauses, but, if we are interested in arriving at a particular conclusion (i.e. we are interested in proving that a particular clause follows from our set of axioms), there is no guarantee that we will come to the one we want to prove. Fortunately, resolution

---

[14]The variables in the two clauses all have to be different. This is always true with clauses since the variables in different clauses are by definition distinct. However, the fact that they may have the same "name" can be confusing! We have provided different names for the variables in the example in order to avoid this problem.

is also *refutation complete*. In other words, if resolution is given a set of inconsistent rules (i.e. at least one of them does not follow from the others) the one and only conclusion it will arrive at is the empty clause, "$\perp$" (i.e. "failure"). Furthermore, it is guaranteed to arrive at that conclusion in a finite number of steps[15]. This is an extremely useful property because now we can use resolution to prove that a predicate **p** follows from a set of axioms by using *refutation*: *given a set of clauses which are consistent,* **p** *is a consequence of them, if we can prove that a new set, formed by including* $\neg$**p** *in the original set, is inconsistent.* This means that, to prove that **p** follows from our set of clauses, we just have to include "$\leftarrow$ **p**" in it, and apply resolution repeatedly. If **p** really does follow from our set of premises the result will eventually be the empty clause "$\perp$". Figure 2-3 shows one way in which resolution would prove that in the example of figure 2-2 John is the grandfather of David.

The problem with resolution, though, is that very often there are many pairs of unifiable complementary terms to choose from at each step, and there is no indication as to which which one(s) should be selected. A sequence of such choices and the associated resolution steps is called a **path**. The set of all the possible paths which can be explored in the search for a solution is called the **search space**. Practical resolution systems often use **heuristics** ("rules of thumb") which help make choices which will lead to a solution faster. These heuristics can be based on parameters such as the number of variables in each goal being considered or the known number of solutions for each goal [20]. In any case, if *all* possible paths are eventually tried (i.e. if the whole search space is explored), and the predicate being proved is actually a *valid conclusion* from the set of axioms, then a solution will eventually be found, although it can take a vast amount of time to do so. On the

---

[15]Provided a *fair* method is used to select clauses. We will return to this point later, in the discussion of Prolog.

• Prove that "`grandfather(john,david)` ← " follows from the set of axioms in figure 2-2. *Resolution proof*:

1. *Add* ← `grandfather(john,david)` *(i.e. the same predicate, but* negated*) to the set of rules*:

```
clause 1: ← grandfather(john,david)
clause 2: father(john,peter) ←
clause 3: father(john,mary) ←
clause 4: father(peter,mike) ←
clause 5: mother(mary,david) ←
clause 6: grandfather(L,M) ← father(L,N), father(N,M)
clause 7: grandfather(X,Y) ← father(X,Z), mother(Z,Y)
```

2. *Resolution step* (clause 1 and 7): *substitutions* X/john, Y/david; *resulting clause*:

```
clause 8: ← father(john,Z'), mother(Z',david)
```

3. *Resolution step* (clause 3 and 8): *substitution* Z'/mary; *resulting clause*:

```
clause 9: ← mother(mary,david)
```

4. *Resolution step* (clause 5 and 9): *substitutions* none; *resulting clause*:

```
clause 10: ⊥
```

• So "`grandfather(john,david)` ← " is proved.

Figure 2-3: Proving that John is the grandfather of David

other hand, if the predicate being proved actually *does not follow* from the axioms, an additional problem arises: since resolution is **only complete for refutations**, then the search is not guaranteed to finish at all! However, despite these limitations, resolution is in practice an extremely useful tool for automated deduction and, as we will see in the next sections, the basis for the concept of programming in Logic.

2.1.3 Horn Clauses

In many applications of logic, it is sufficient to restrict the set of possible representations of clauses to those with at most one conclusion. Clauses in this form are called **Horn Clauses** and they are just a special case of clausal form: a *Horn clause is a clause which has only 0 or 1 atomic predicates to the left of the implication* ($\leftarrow$). It has been proved that any problem which can be expressed in logic can be expressed using the Horn clause formalism [40]. As an example, all the clauses in figure 2-3 are Horn clauses.

2.2 Logic as a Programming Language

In the previous sections we have presented symbolic logic predicates as an elegant and concise way of expressing knowledge, and clausal form as a simple way of writing these predicates. We have also shown how a simple inference rule, resolution, can be used to infer conclusions from this knowledge which are logically sound.

Resolution appears to be a very attractive idea, not only for finding conclusions in logic systems, but also for **solving a more general class of computational problems**. The simplicity and iterative nature of resolution make it possible to take advantage of the expressive power of logic, while keeping simple procedural semantics which are suitable for computer implementation. Thus, the idea arises of *using Logic as a Computer Programming Language*. The advantages of such a choice were discussed in Chapter 1. Logic programs are usually written using Horn clauses, because resolution with Horn clauses is relatively simple. Unless otherwise noted, from now on the term "Logic Program" will refer to a Horn clause program.

2.2.1 Syntax of Horn Clause Programs

A **logic program** is a set of *Horn clauses*. A horn clause is an expression of the form[16]:

`head :- goal`$_1$`, ..., goal`$_n$`.`

where the only conclusion ("`head`") is called the **head** of the clause and the conditions "`goal`$_1$`, ..., goal`$_n$", ($n \geq 0$) are called the **body** of the clause. All `head`, `goal`$_1$`, ..., goal`$_n$ are *atomic predicates*, as defined previously. The predicates in the *body* are also called **goals** or **procedure calls**. If `n=0` (i.e. the body is empty) the clause is called a **fact**, and the ":-" symbol is omitted. Otherwise the clause is said to be a **rule**. A *headless clause* is called a **query**. A set of clauses whose *heads* all have the same *predicate symbol* and *arity* is called a **procedure** or a **relation**. The *terms* in an atomic predicate are also called its **arguments**.

An example will make these definitions more clear: figure 2-4 shows the family relationship example of figure 2-2 written in this syntax. "`grandfather(L,M)`" is the *head* of one of the clauses of the "`grandfather`" *procedure*. Its *arguments* are the variables "`L`" and "`M`". *Goals* "`father(L,N)`, `father(N,M)`" form its *body*. The "`grandfather`" *procedure* itself comprises the two "`grandfather`" clauses.

---

[16]Note that this notation is identical to the one given for the general form of clauses, but using simpler symbols for those which are awkward to represent in a computer (for example ← is replaced by ":-"). Also, some new terminology such as **head**, **body**, and **goals** etc. is introduced in order to more easily refer to the different parts of the clause.

```
father(john,peter).
father(john,mary).
father(peter,mike).

mother(mary,david).

grandfather(L,M) :- father(L,N), father(N,M).
grandfather(X,Y) :- father(X,Z), mother(Z,Y).
```

Figure 2-4: A Family Relationship Logic Program

2.2.2 Declarative Semantics

The declarative reading[17] of the clauses in a logic program derives directly from the one given for the general clause form. Recalling the assumptions made therein, if the *Horn* clause

$\texttt{head :- goal}_1\texttt{, ..., goal}_n\texttt{.}$

contains the variables X_1, ..., X_k, then the expression above is a shorthand for

$\forall\ \texttt{X}_1\texttt{, ..., } \forall\ \texttt{X}_k\ \texttt{head} \leftarrow \texttt{goal}_1 \wedge \ldots \wedge \texttt{goal}_n$

therefore, a **rule** ($\texttt{n} > \texttt{0}$) can be read as

for all X_1, ..., X_k,
head
if $goal_1$ *and ... and* $goal_n$

and a **fact** ($\texttt{n=0}$) can be read as

for all X_1, ..., X_k,
head is always true.

A **query** (the headless clause) can be read as

For which X_1, ..., X_k

---

[17] Model-theoretic and fixpoint declarative semantics have been also studied, see [83].

are $body_1$ *and ... and* $body_n$ *always true?*

Also, if `m=n=0` then the clause represents the predicate that is always false, and it can be read as **"fail"**.

2.2.3 Procedural Semantics

The execution of most logic programming systems is based on **top-down** (or "goal oriented") resolution[18] , which will be introduced shortly. Consistent with the *refutation* method used in any resolution proof, if the question to be answered is whether the fact

$query_1$, ..., $query_n$.

is true, the actual *query* posed to the system would be the same fact, but negated:

?:- $query_1$, ..., $query_n$.

(the "?" mark is included in front of the query to emphasize that it is actually a question). Execution of a program is invoked by this query, and it amounts to a series of *top-down resolution steps*, also called **top-down derivations** or **computations**. Each such step comprises the following actions[19]:

1. *Select one of the procedure calls in the query.*

   - *If there are none, exit, and report "*`YES.`*" (***success***). If variables in the query have been bound (substituted), report the substitutions.*

2. *Find the clauses whose head will unify with the selected call.*

   - *If none are found, exit, and report "*`NO.`*" (***failure***).*

3. *Select one of those clauses for the next step.*

4. *Apply resolution to the selected clause and procedure call. A new*

---

[18] Bottom up systems have also been proposed, but are to date less efficient than top-down systems, specially in the presence of recursion [4].

[19] Note that this is the basic resolution algorithm, but with some "built-in" heuristics: for example, always using the *most current* clause generated (starting with the query) as one of the two clauses being resolved. This algorithm is, however, still "refutation complete".

(headless) clause is obtained which is basically the query, but with the body of the selected clause in place of the unified procedure call, and the variable substitutions resulting from unification extended to the whole clause.

5. *This new clause is considered the new query, and execution continues at 1 above.*

As seen above, the process continues until the query is reduced to the empty clause (and **"success"** is reported), or until no head can be found that matches any of the calls in the query (and the reported result is **"failure"**). The example in figure 2-3 happens to follow top-down resolution, and is therefore one of the possible executions of a logic program whose sentences are those in figure 2-4 in response to the query `"?:- grandfather(john, david)."`. Other possible queries to that same program, and the responses obtained by applying the above algorithm are offered in figure 2-5. As in the general resolution algorithm, each possible set of steps leading from a query to a solution is called an *execution path*, and all possible paths which can be explored while looking for an answer to a given query form the **search space** for that query and for that particular program.

2.2.4 Non-Determinism and the Control Strategy

It should be fairly clear from the description above that there are two distinct components during the execution of a Logic Program:

1. The **program**, i.e. the set of rules and facts, provided by the user (including the *query*).

2. An **evaluator** of the program, which is in charge of answering the query using the *top-down resolution* algorithm given above.

It should also be clear from that description that there are two occasions (steps 1 and 3 in the *top-down resolution* algorithm) in which the next step to be

Query/System Response:	Translation:
`?:- father(john,peter).` `YES.`	"Is John the father of Peter?"
`?:- father(X,mike).` `X=peter.`	"Who is the father of Mike?"
`?:- grandfather(john,mike).` `YES.`	"Is John the grandfather of Mike?"
`?:- mother(mary,peter).` `NO.`	"Is Mary the mother of Peter?"
`?:- grandfather(john,W).` `W=mike.`; also `W=david.`	"Who is John the grandfather of?"

Figure 2-5: Results of Top-down Resolution for the Program in Figure 2-4

taken by the program *evaluator* is not uniquely determined. This is the origin of the two basic types of non-determinism present in Logic programs [40]:

- **non-determinism$_1$**: if several clause heads unify with the selected goal, step 3 does not *determine*$_1$ which of them is to be selected. The *policy* used by the program evaluator for performing this selection is called the **search rule**. The *search rule* also determines whether the remaining choices will also be eventually tried or not. This results in two subtypes of *nondeterminism*$_1$:
  - **"Don't care" non-determinism$_1$**: once a choice is made the system *commits* to that choice.
  - **"Don't know" non-determinism$_1$**: more than one of the possible choices may eventually be tried in the search for a solution.
- **non-determinism$_2$**: if the current *query* contains several goals (procedure calls) step 1 does not *determine*$_2$ which one of them will be selected for execution next. The *policy* used by the program evaluator for performing this selection is called the **computation rule**.

The *search rule* and the *computation rule* together define the **control strategy** used by the evaluator. It is important to note that modifying the *search rule* affects the order and number of *solutions* which can be obtained from the system: although top-down resolution does not impose a particular order in the choices made by the *search rule*, *completeness* (i.e. the guarantee of finding all possible solutions) is only preserved if a **fair rule** is chosen, i.e. one which will assure that all possible paths in the *search space* will eventually be explored. Systems which use only *"don't care" non-determinism*$_1$ are therefore *incomplete*. Furthermore, they can only provide at most one solution path for a given query. Systems which use *"don't know" non-determinism*$_1$ can provide more than one solution to a given query. Their degree of *completeness* depends on the type of *search rule* being used. Since most *computation rules* are **exhaustive** (i.e. they will eventually invoke all goals in the body of a clause) the choice of one or another will only affect the behavior of the system, but not the number of solutions found.

2.2.5 The AND/OR Tree Representation of the Search Space

The different *execution paths* which are possible for a given query and program (i.e. the *search space*) are often represented pictorially in the form of an (inverted) *AND/OR tree*. The root of the tree is the query and each branch is a top-down derivation. Each node represents an application of either the *search rule* (**"OR-nodes"**) or the *computation rule* (**"AND-nodes"**). Consistent with the exhaustive computation rules used in Logic Programs, all possible paths under an *AND-node* have to be explored. This is represented by linking the branches with an arc. On the other hand, the paths which are explored under an *OR-node* and the order in which it is done *depend on the search rule being used*. Figure 2-6 is an *AND/OR tree* representation of the search space for the query and program of figure 2-3.

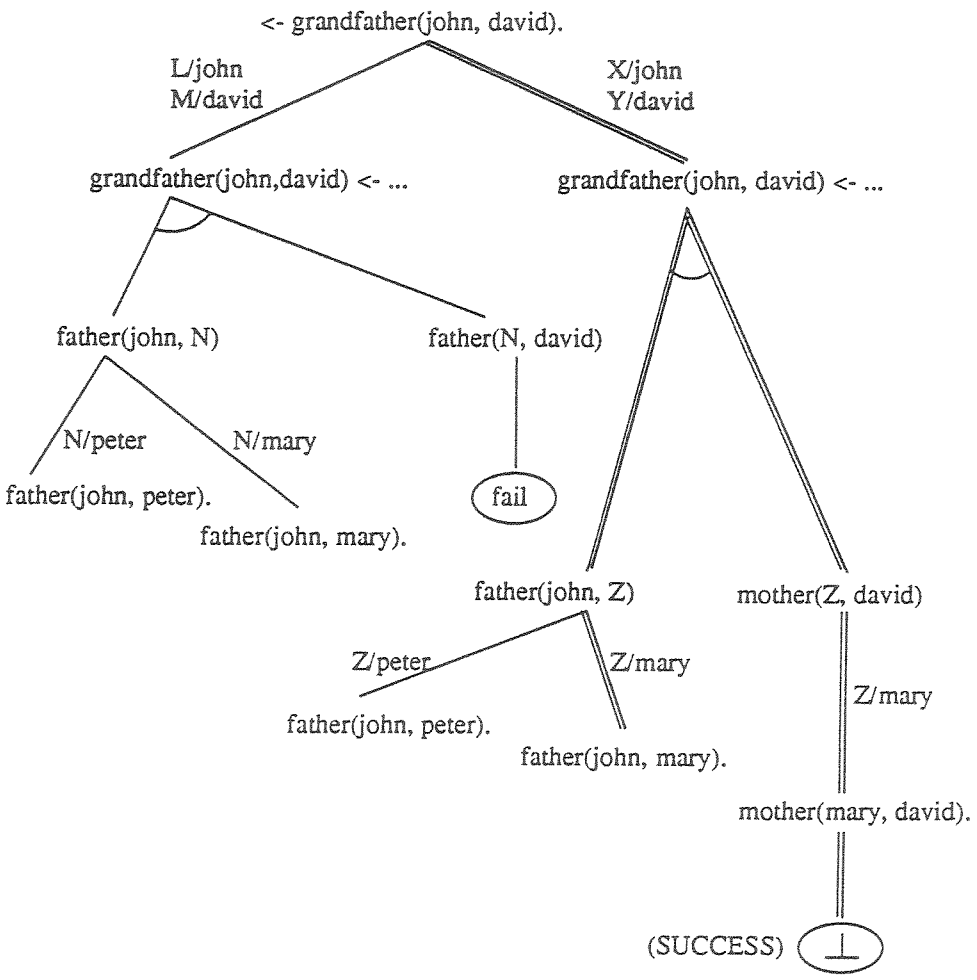

Figure 2-6: AND/OR Tree Representation of a Search Space

2.2.6 The Logical Variable

One of the most characteristic features of Logic Programming is the behavior of the Logical variable. For example, it exhibits *bidirectionality*, based on the bidirectionality of unification: the same variable in the same clause can serve as either an input or an output parameter depending on what it is being unified against. For example, when the query `"?:- grandfather(john,david)."` in figure 2-5 is unified with the head of the clause

```
grandfather(L,M) :- father(L,N), father(N,M).
```

M acts as an *input* variable, which conveys information from the query to the clause. On the other hand, when answering the query "`?:- grandfather(john,W).`", M acts as an *output* variable, conveying information (the answer `W=mike.`) from the clause to the query. Thus, the same procedure can be executed with different patterns of input and output parameters. This also means that (pure) Logic Programs, if correctly stated, can be run "backwards" producing their inputs from their outputs: a procedure to perform square roots can be used to square numbers, and procedures which compute derivatives can be used to compute integrals [6].

Another peculiar characteristic of the logical variable is that a procedure can construct structures containing variables which can be "filled in" later by other procedures. These structures are said to be *partially instantiated.* This technique has proved very useful and is widely used in logic programming practice[20].

2.2.7 Transparent Control

Theoretically, in a logic programming system the programmer does not need to be concerned with the *control strategy* used by the program evaluator. Provided a *control strategy* which preserves *completeness* is used, the questions which are posed to the system are guaranteed to be correctly answered by the program evaluator. We refer to such a system as a *transparent control* system: the programmer only needs to provide a list of axioms (knowledge) about a particular problem containing enough information to solve it, and the program evaluator will "do the work". This view of logic programming is clearly very appealing, and it can actually be implemented in relatively simple applications, such as in database query.

---

[20]For example, Warren has used it for compiler writing in Prolog [86] and Gregory [30] and Shapiro [72] in the "incomplete message" communication technique often used in their parallel languages.

For most applications, though, it would be necessary to devise a far more sophisticated *control strategy* than any of those known today to be able to solve any substantial problem from only its declarative description in a reasonable amount of time. Therefore, practical systems offer the programmer means to *affect* the control strategy used by the program evaluator in various ways. For example, the search rule can be forced to explore some candidate clauses before (or instead of) others, perhaps because it is known that they are more likely to find a solution. This *control* information can be provided *explicitly* through *annotations*. Annotations can appear embedded within the original clauses or in a separate list. The set of possible annotations is called the **control language**. Alternatively, control information can be expressed *implicitly*, in the ordering of the clauses within the program, and in the ordering of the goals within the body of a clause.

2.2.8 Prolog

Prolog[21] (PROgramming in LOGic) was the first practical logic programming language and it still is the most widely used and efficiently implemented today. It was devised by the group led by A. Colmenauer at the U. of Marseille. They chose for Prolog an extremely simple *implicit control strategy*, based on the ordering of clauses within the program, and on the ordering of the goals in the bodies of the clauses. The following two rules determine Prolog's *control strategy*:

- *Search rule*: given a goal, the first clause whose head unifies with the goal, scanning from top to bottom of the program, is selected. Then the goals in the body of the clause are executed in the order determined by the *computation rule* below. If the choice does not lead to a solution (i.e. it leads to "failure"), all resolution steps and variable substitutions (i.e. all "bindings") done since the last such choice are undone, the next clause whose head matches with the goal is selected, and execution continues from there. This technique is called **backtracking**.

[21] We only have space for outlining the basic elements of the language here. A more detailed description can be found in [16], or in the DEC-10 Prolog [58] or Quintus Prolog [64] user's manuals, which are some of the fastest, and most widely used current implementations.

- *Computation rule*: once a clause is selected (using the *search rule* above), the goals in the body of the clause are executed one by one in left-to-right order.

This control strategy is called **depth-first search with backtracking** and its main advantages are its simplicity and its potential for very efficient implementation. Prolog supports *"don't know" non-determinism* through *backtracking*. It also provides a mechanism (the *"cut"*, to be explained later) for achieving *"don't care" non-determinism* when needed. Figure 2-7 shows a trace of the execution of Prolog while evaluating the query "`?:- grandfather(john,X).`" in the family relation program of figure 2-4.

As a further example, the following Prolog program will check whether a number is a positive integer or not:

```
is_pos_integer(0).
is_pos_integer(X):-is_pos_integer(Y), X is Y+1.
```

The predicates "`is`" and "`+`" are built into the language. This program has a nice declarative interpretation: it defines an integer recursively as either zero, or another integer incremented by one. This happens to be an almost formal definition of integer numbers! Interestingly, the program can also be used to *generate* positive integers: the response to the query

```
?is_pos_integer(X).
```

is 0, and then 1,2,3,4... : the list of all positive integer numbers.

Prolog offers several *built-in predicates* (such as "`is`" and "`+`") which make the task of programming problems other than simple database queries (i.e. those illustrated thus far) somewhat easier. Some of these predicates can be expressed in terms of the basic language and are there only for convenience. For example, lists can always be written as compound terms (structures): the list whose elements are

Program:

```
father(john,peter).
father(john,mary).
father(peter,mike).
mother(mary,david).
grandfather(L,M) :- father(L,N), father(N,M).
grandfather(X,Y) :- father(X,Z), mother(Z,Y).
```

Query:

```
?:- grandfather(G,david).
```

Execution Steps (current state:)	*Next step; variable bindings*
Step 1: ?:- grandfather(G,david)	select 1st. "grandfather" clause which unifies: grandfather(L,M)...; L/G, M/david
Step 2: father(G,N), father(N,david)	select leftmost goal, select 1st."father" clause which unifies; G/john, N/peter
Step 3: father(peter,david)	no clause unifies with this one, FAIL: return to the last choice point (step 2); undo G/john, N/peter
Step 4: father(G,N), father(N,david)	select leftmost goal, select next "father" clause which unifies; G/john, N/mary
Step 5: father(mary,david)	no clause unifies with this one, FAIL: return to the last choice point (step 2); undo G/john, N/mary
Step 6: father(G,N), father(N,david)	select leftmost goal, select next "father" clause which unifies; G/peter N/mike
Step 7: father(mike,david)	no clause unifies with this one, FAIL: return to the last choice point (step 2); undo G/peter, N/mike
Step 8: father(G,N), father(N,david)	there are no more "father" clauses, FAIL: return to last choice point (step 1), undo L/G, M/david
Step 9: ?:- grandfather(G,david)	select next "grandfather" clause which unifies: grandfather(X,Y)...; X/G, Y/david
Step 10: father(G,Z), mother(Z,david)	select leftmost goal, select 1st."father" clause which unifies; G/john, Z/peter
Step 11: mother(peter,david)	no clause unifies with this one, FAIL: return to the last choice point (step 10); undo G/john, Z/peter
Step 12: father(G,Z), mother(Z,david)	select leftmost goal, select next "father" clause which unifies; G/john, Z/mary
Step 13: mother(peter,david)	this fact is unified, no bindings made.
Step 14: $\perp$	SUCCESS: report query bindings: "G=john"

Answer:

```
G = "john"
```

Figure 2-7: Prolog Execution of the Family Relationship Program

`a,b,c,d` can be represented as the structure `.(a,.(b,.(c,.(d,[]))))`, where the constant `"[]"` represents the empty list. Prolog, however, provides a more compact notation: the list above can be represented in Prolog as `[a,b,c,d]`. Also, in Prolog, the notation `[X|Y]` represents the list whose *head* (or "first element") is `X` and whose

tail (or "rest of the elements") is `Y`. Thus, if we unify `[a,b,c,d]` with `[X|Y]` the resulting substitutions will be `X/a` and `Y/[b,c,d]`. Several other built-in predicates (their syntax often a function of the particular implementation) support more conventional computer language features such as integer arithmetic, input/output, file access, success and failure, term classification, and data structures. Often there are also debugging facilities.

Some of the built-in predicates of Prolog and the general programming style will be introduced in the following examples. As a first example, we will take the problem of *appending* two lists, i.e. a program which will answer the query

```
?:- append([a,b],[1,2,3],Result).
```

with

```
Result=[a,b,1,2,3]
```

The following is an "append" program written in Prolog:

```
append([],List,List).
append([Head|Tail],List2,[Head|Tailandlist2]):-
                append(Tail, List2, Tailandlist2).
```

The declarative reading is clear[22]:

- *Clause 1:* The result of appending the empty list to a list is the same list.
- *Clause 2:* The result of appending List1 (i.e. "[Head|Tail]") and List2 is a list whose *head* (first element) is the same as the head of List1, and whose *tail* is the result of appending the tail of List1 to List2.

---

[22] A more "intelligible" way of writing this program (though less efficient) is:

```
append([],List,List).
append(List1, List2, Result):-
                [Head|Tail] = List1,
                append(Tail, List2, Tailandlist2),
                Result = [Head|Tailandlist2].
```

Procedurally, execution of the query above would start by unifying this query with the head of the first clause (`Head/a, Tail/[b], List2/[1,2,3], Result/[a|Tailandlist2]`). Note that the result is starting to be **cons***tructed* as the list `"[a | Tailandlist2]"`, with `Tailandlist2` still a (free) variable. The next step (body of the first append clause) is to call "append" (recursively) with the appropriate substitutions:

```
:- append([b], [1,2,3], Tailandlist2).
```

This new query unifies again with the first "append" clause. The substitutions are now `Head'/b, Tail'/[], List2'/[1,2,3]` and `Tailandlist2` will be *constructed* as `"[b | Tailandlist2']"` where `Tailandlist2'` is a new (free) variable. The next step calls "append" again with the appropriate substitutions:

```
:- append([], [1,2,3], Tailandlist2').
```

which unifies with the second clause (`[]` does not unify with `[X|Y]` because it is a constant, being unified with a list -i.e. a compound term) with the substitution `Tailandlist2'/[1,2,3]` and we finally succeed. Since we have gathered the substitutions

```
Result=[a | Tailandlist2]
Tailandlist2=[b | Tailandlist2']
Tailandlist2'=[1,2,3]
```

we can report the result

```
Result=[a,b,1,2,3]
```

The following program works in a very similar way to append:

```
split([A|X],Pivot,Y,[A|Z]) :- A =< Pivot, split(X,Pivot,Y,Z).
split([A|X],Pivot,[A|Y],Z) :- A >  Pivot, split(X,Pivot,Y,Z).
split([],_,[],[]).
```

it splits a list into two lists, one containing all elements which are $\leq$ than the constant in `Pivot`, and the other containing the rest of the elements. Thus, the answer to

```
?:- split([1,5,7,6,3,2,9], 5, Biglist, Smalllist).
```

would be

```
Biglist=[7,6,9]
Smalllist=[1,3,2]
```

The first clause is tried first. If `A =< Pivot` succeeds, then `A` is made part of `Smalllist`. If the test fails, then the other clause is tried and the result is made part of `Biglist`.

With the aid of the two procedures defined above, we can write a simple "quicksort" [33] algorithm:

```
qsort([],[]).
qsort([Pivot|Rest],Orderedlist) :- split(Rest,Pivot,Big,Small),
              qsort(Big,   Sortedbig),
              qsort(Small, Sortedsmall),
              append(Sortedbig, [Pivot|Sortedsmall], Orderedlist).

split([A|X],Pivot,Y,[A|Z]) :- A =< Pivot, split(X,Pivot,Y,Z).
split([A|X],Pivot,[A|Y],Z) :- A >  Pivot, split(X,Pivot,Y,Z).
split([],_,[],[]).

append([Head|Tail],List2,[Head|Tailandlist2]):-
              append(Tail, List2, Tailandlist2).
append([],List,List).
```

which will provide answers such as

```
?:- qsort([3,6,7,2,1], Orderedlist).

    Orderedlist=[1,2,3,6,7]
```

The "quicksort" algorithm constructs an *ordered* copy of a given list by taking the first element of the list (the *Pivot*) and splitting the rest of the list into two, one with elements which are "bigger" and another one with elements which are "smaller". If the same algorithm is repeated for the new lists obtained until only empty lists remain the result is an ordered version of the original list.

As mentioned before, Prolog also offers a way of controlling backtracking, which is basically an implementation of *"don't care" non-determinism*: the **"cut"** (!). "Cut" basically *commits the system to all the choices made since the clause in which the* **cut** *is encountered was called.* Cut is often used by programmers for example to eliminate alternate execution paths when it can be determined that the solution obtained thus far is valid and there is no need for another one. It is also used to limit memory use in certain implementations because it frees resources and to avoid possible loops due to the depth-first search procedure. Another interesting feature is the use of **assert** and **retract**. Using them, clauses can be included or taken out of the program dynamically. Finally, a number of *meta-level* predicates are included which can construct clauses, or read parts of the program as data. These facilities make it possible to write *meta-level* interpreters [17] in Prolog.

Side-effects such as *cut*, *assert* and *retract* can be very useful in practice, but they are fairly controversial because of their harm to the declarative semantics of the language: programs which contain these side effects can only be understood by referring to procedural semantics. Unfortunately, there are often cases where even a correctly written declarative description of a problem using no side effects will not give the expected results when executed by a Prolog system. This is due to the fact that the depth-first search control strategy used in Prolog *does not achieve completeness*: it is not *fair*, because the top clauses in the program are always tried first. If the search space is *infinite* (which is often the case if there are recursive rules in the program) some clauses may never be reached. This can lead to loops, so that attention to procedural semantics, and careful ordering of the clauses (and, sometimes, even the use of *cut*) are often needed to make a program run correctly. *Correctness*, however, is still achieved: i.e. all solutions obtained from a (side effect free) Prolog program correspond to its declarative semantics, although not all of the possible solutions may be reached.

Despite its *incompleteness*, Prolog is in practice an extremely useful and efficiently implemented programming language [76]. It was already pointed out how most practical Logic Programming systems trade completeness for efficiency. In this sense it is interesting to note that while many proposals for new Logic Programming languages attempt to achieve "more completeness" than Prolog, many others are purposedly "less complete" (in the sense that they would obtain a smaller set of answers than Prolog), their incompleteness again being defended with implementation efficiency arguments. This is the case, for example, of systems which only support *"don't care" non-determinism*. In *sequential* systems Prolog then seems to offer still today a compromise position which has made it useful in applications such as relational databases, mathematical logic, abstract problem solving, natural language understanding [61] , architectural design, symbolic equation solving, plane geometry, learning, planning, robotics, compiler design, and in many other areas not limited to Artificial Intelligence.

2.3 Chapter Summary

This chapter has dealt with well established concepts in Logic Programming. The clausal form for Symbolic Logic was introduced and Resolution presented as a simple inference rule capable of proving theorems by refutation. It was also shown how this same mechanism could be used to answer questions by inferring the correct answers from a set of axioms. This led to the idea of Programming in Logic. The declarative and procedural semantics of Horn clause Logic Programs were then presented pointing out the separation of the program from the control strategy. The Language Prolog was then introduced as an example of a practical Logic Programming Language with a particular control strategy. Other control strategies leading to the idea of executing Logic Programs in parallel will be presented in the next chapter.

Chapter 3

Parallelism and Logic Programs

This chapter deals with the relationship between Logic Programming and parallelism. Based on the definition of top-down resolution, and the syntax and semantics of Logic Programs offered in Chapter 1, the different sources of parallelism offered by the procedural interpretation of Logic are introduced. The problems associated with the implementation of some of these sources in practice are then discussed. Finally, a general approach is proposed for the implementation of Logic Programs in parallel which will attempt to solve such problems in an efficient way. This approach will guide the design of the execution model of the following chapters.

3.1 Parallelism in Logic Programs

It was already mentioned in Chapter 1 how the renewed interest in massively parallel architectures and *the realization of the complexity involved in programming such architectures* has spurred increasing attention to Logic Programming languages (and other declarative formalisms) and their computational models. The relationship between Logic Programming and parallelism is based on the "freedom" (non-determinism) which the program *evaluator* has in choosing execution paths: a possibility which remains open in the formulation of *resolution* is *executing several of those paths in parallel*. There are two basic lines of reasoning which make it an attractive idea to exploit this potential parallelism.

From the *language research point of view* an area of interest is to approach

the initial goal of preserving the declarative nature of Logic in logic programming languages. In order to achieve this goal, "features" which can only be explained through *procedural* considerations have to be avoided. A *resolution complete* search strategy, more sophisticated than Prolog's simple depth-first search with backtracking, is required to achieve such behavior. One of the reasons why parallelism comes into play in this area is that many *resolution complete* search strategies often lend themselves naturally to parallel execution.

From the *applications point of view*, one of the most compelling issues is, of course, performance. In order to meet the requirements of many present and future applications, higher inference speeds will be needed than those which can be reasonably expected from a *sequential* von-Neumann machine using today's technology or that of the near future. The opportunity for parallel execution offered by Logic Programs can be used to exploit the performance potential of new, parallel architectures.

3.1.1 Sources of Parallelism

As mentioned above, the two basic types of non-determinism present in the procedural interpretation of a Logic Program as a top-down resolution proof are also the origins of the two main sources of parallelism present in Logic Programs [19]:

- **OR-Parallelism:** *a* process *can be assigned to solve the body of every clause that is active, i.e. every clause whose head unifies with a given goal (***non-determinism**$_1$*).*

- **AND-Parallelism:** *a* process *can be assigned to solve each of the goals in the body of an active clause (***non-determinism**$_2$*).*

In the above definitions a **process** can basically be viewed as an independent Logic Program *evaluator*. Note how *AND-* and *OR-Parallelism* refer to the parallel exploration of the paths under the AND and OR nodes of the AND-OR tree representation of the search space mentioned in the previous chapter.

In addition to the two basic types of parallelism presented above, other lower-level types of parallelism which are not based on non-determinism are also possible:

- **Stream Parallelism:** *several processes can evaluate complex data structures incrementally, in parallel with the process which is producing them.*

- **Search Parallelism:** *the program can be divided into disjoint sets of clauses so that several processes can search for clauses whose heads unify with a given goal in parallel, each working on a different set.*

Yet another possible source of parallelism is

- **Unification Parallelism:** *when unifying a goal with the head of a clause, several pairs of corresponding terms (arguments) can be unified in parallel.*

3.1.2 An Example Showing Different Types of Parallelism

In order to better understand the definitions of the previous section, we will follow the execution of a simple example showing the points where parallel execution is possible. For the rest of this section we will refer to the Logic Program in figure 3-1 which represents a simple database. Suppose the following query is posed to the system:

```
?:- crew(Member1, luis, boing77).
```

i.e. we are trying to set up a a suitable crew for a "boing77" aircraft, and we are asking the system to find somebody which can be the first member of the crew, and to check whether `luis` can be the other member. The only candidate clause for unification with this query is the "crew" clause. Note that the actions of unifying `Member1` with `X`, `luis` with `Y`, and `boing77` with `boing77` are all independent and can be done in parallel (*unification parallelism*). Since this unification succeeds, the query is reduced to

```
:- qualified(Member1, pilot), qualified(luis, radio_operator).
```

These two problems (finding a qualified pilot, and checking whether `luis` is a qualified radio operator) are also *independent*. Therefore, a different process can be in charge of each of these tasks (*(goal independence) AND-Parallelism*).

```
crew(X, Y, boing77):- qualified(X, pilot), qualified(Y, radio_operator).

qualified(X, pilot):- has_license(X, type(civil)),
                      has_medical(X, classIII).
qualified(X, pilot):- has_license(X, type(military)), has_rating(X, civil),
                      has_medical(X, classIII).

has_license(ian,    type(civil)).        has_medical(sabina,classIII).
has_license(pat,    type(civil)).        has_medical(pat,   classII).
has_license(sabina, type(civil)).        has_medical(manuel,classIII).
has_license(lola,   type(civil)).        has_medical(lola,  classI).
has_license(manuel, type(civil)).        has_medical(ian,   classIII).

has_license(javier, type(military)).     has_medical(javier,classII).
has_license(jaime,  type(military)).     has_medical(jaime, classIII).
has_license(luis,   type(military)).     has_medical(luis,  classI).
has_license(yayo,   type(military)).     has_medical(yayo,  classIII).
has_license(felipe, type(military)).     has_medical(felipe,classI).

has_rating(javier,  civil).
has_rating(jaime,   civil).
has_rating(yayo,    civil).

qualified(Z,radio_operator):- misc_qual(Z,Qual_list),
                              is_in_list(radio_operator, Qual_list).

misc_qual(ian,     [radio_operator, mechanic, navigator, scuba_diver]).
misc_qual(luis,    [mechanic, navigator, instructor, radio_operator]).
misc_qual(felipe,  [instructor, mechanic]).

is_in_list(Element, [First_element | Rest]):- Element = First_element.
is_in_list(Element, [First_element | Rest]):- is_in_list(Element, Rest).
```

Figure 3-1: Fictional Aviation Administration's (FAA) Database

Let us first follow the execution of the process in charge of solving the leftmost goal:

```
:- qualified(Member1, pilot).
```

This goal matches two clauses in the program,

```
qualified(X, pilot):- has_license(X, type(civil)),
                      has_medical(X, classIII).
qualified(X, pilot):- has_license(X, type(military)),
                      has_rating(X, civil),
                      has_medical(X, classIII).
```

describing two ways the qualifications required for flying civil aircraft can be met. Clearly, two independent processes can be used again: one looking for civilian pilots (executing the first clause) and the other one looking for military pilots which have obtained a civil rating (second clause). This is an example of *OR-Parallelism*.

The first one of the above mentioned processes would be in charge of solving `:- has_license(Member1, type(civil)), has_medical(Member1, classIII).` Note that now the two goals in the body of the cause *are not independent*. We cannot simply go ahead and evaluate `"has_license(Member1, type(civil))"` and `"has_medical(Member1, classIII)"` in parallel, because they will independently generate a value for `"Member1"` but both values might not be the same, and this is required by the semantics of the clause. For example, `"has_license(Member1, type(civil))"` could find `Member1/Ian` and `"has_medical(Member1, classIII)"` could find `Member1/Sabina`. This is called a **variable binding conflict**, one of the problems posed by AND-Parallelism in practice. The simplest way of dealing with this *variable binding conflict* is to execute the goals sequentially, one after the other: first a solution is found for

```
:- has_license(Member1, type(civil)), ...
```

using the first `"has_license"` clause. The substitution obtained (`Member1/ian`) is then propagated, so that the next goal is now simply

```
:- has_medical(ian, type(civil)).
```

Now it is easy to find the clause which unifies with this goal (the fifth clause of the `"has_medical"` procedure). However, in order to show the usefulness of

search parallelism, suppose that the number of clauses in this procedure were large: then, it might take a long time to explore all the possible clauses looking for the appropriate one. An alternative would be to divide the procedure into several sets of clauses, and use a different process to scan each set independently for the the sought for clause. This is an example of *search parallelism*.

Let us now return to the AND-Parallel process which was left in charge of finding whether `luis` is a qualified `radio_operator` at the first clause in the program. This process would be in charge of the execution of

```
:- qualified(luis, radio_operator).
```

which is reduced to

```
:- misc_qual(luis,Qual_list), is_in_list(radio_operator,Qual_list).
```

A *variable binding conflict* can also appear in this clause, since the variable `Qual_list` is common to both goals. The same technique used before (sequential execution) could be applied here. However, this time a different technique (*stream communication*) will be used. Note that the purpose of calling `"misc_qual(luis,Qual_list)"` is to *produce* a *list* of qualifications for `luis`, and this is done during the unification of the call with

```
misc_qual(luis,    [mechanic, navigator, instructor, radio_operator]).
```

The purpose of `"is_in_list(radio_operator, Qual_list)"`, in turn, is to look at the elements of the list one by one and check if `radio_operator` is in it. In the *sequential execution* approach this would be done in two sequential steps: first `"misc_qual"` would produce the entire list, and then, once complete, `"is_in_list"` would check it. The alternative is to start both goals in parallel, so that as `"misc_qual"` produces each element of the list of qualifications it is immediately passed on to `"is_in_list"` which checks it. In this mode of operation there is a *stream* of elements passing from one goal to the other (*stream*

AND-Parallelism). In general, there can be several *consumers* of such a shared variable, but only one *producer*. Note that if the variable being shared is not a compound term (list or structure) then execution is actually sequential. Stream AND-Parallelism is similar to *pipelining*.

An alternative to stream AND-Parallelism is to also start both goals in parallel, but make the consumer wait until the shared variable is fully instantiated: i.e. in the previous example `"misc_qual"` and `"is_in_list"` would be started in parallel, but `"is_in_list"` would wait until the list of qualifications is completely constructed. This technique of goal *suspension* is, however, in this simple example, roughly equivalent to sequential execution.

3.2 Logic Programs and Parallelism in Practice

It should be clear from the previous sections that logic programs offer many sources of parallelism. One problem which arises in practice is that of detecting this potential for parallelism in a given program. There are at least two ways of performing this detection: potential parallelism can (at least theoretically) be uncovered and managed automatically by the program evaluator, aided perhaps by some compile-time analysis. This has the advantage of relieving the programmer from keeping track of communication, synchronization and concurrency issues, and makes it possible to improve performance by adding resources in a user-transparent way. Alternatively, the responsibility of uncovering this parallelism can be put in the hands of the programmer, by extending the *control language* (i.e. that in charge of specifying the control component in a logic programming language using *explicit control specification*) to include constructs or annotations which explicitly invoke and handle parallel execution.

Another issue which has to be addressed is the types of parallelism which are to be exploited in a given system. Ideally, all possible sources should be taken advantage of. However, the management and control of this parallelism is non-trivial and the overhead involved in exercising these management functions could completely overshadow any performance gains obtained through parallel execution. If, as is often the case, efficiency is an important issue in the design, due consideration has to be given to the run-time cost associated with the implementation of the different types of parallelism which are chosen to be supported. Also, certain types of parallelism present serious implementation problems in practice such as the *variable binding conflicts* which we already encountered in AND-Parallelism.

In the following sections we will address some of the problems associated with parallel Logic Programming implementation, review some of the techniques which have been proposed in order to deal with these problems, and consider the *cost* associated with the implementation of such techniques in practice. Attention will be given primarily to *AND-* and *OR-Parallelism*. Although they are also interesting sources of parallelism, we will not address *search-* and *unification-parallelism* in this chapter, because their implementation is generally done at a lower level in the design and because this implementation is usually fairly independent from that of *AND-* and *OR-Parallelism*[23]. *AND-* and *OR-Parallelism* can be combined in practice in several ways giving rise to a number of different *forms of parallelism* [30] which will also be addressed in the following paragraphs.

---

[23]Search parallelism is a very promising source of performance improvement, but even at the lowest level it will be outside the scope of this dissertation: we will model search parallelism as concealed within the context of the clause indexing mechanism of the abstract machine which will be introduced in Chapter 7.

3.3 Pure OR-Parallelism

It was already pointed out in Chapter 1 how the implementation of **OR-Parallelism** is, at least in principle, relatively straightforward, since the parallel processes involved are fully independent. OR-Parallel systems usually rely on either passing independent copies of the complete state of the computation up to the branching point to each of the alternative paths to be evaluated in parallel, or on keeping local copies of only the parts of the environment which are to be written by alternate clauses, while other parts are shared.

OR-Parallel execution generally implies a *search rule* which is *fair*, since all possible paths are eventually tried. This *search rule* can be combined with a sequential *computation rule* (such as that of Prolog) or with any of the AND-Parallel schemes which will be described in the next sections. Because of the *fairness* of a parallel search rule, OR-Parallelism is generally "more complete" than sequential *depth-first* systems (such as Prolog). However, full application of OR-Parallelism at each possible branching point in the computation suffers from the general inefficiency of any *complete* system: it can result in a combinatorial explosion in the size of the search space to be explored, and in the number of processes generated. This is aggravated by the fact that sometimes, if only one solution is needed, much of the computation cannot be considered "useful work". Solutions proposed for this problem include the use of annotations in order to restrict the generation of OR-Parallel alternatives [55] and the use of heuristics in order to prune as many of the paths not leading to a solution as early as possible in the computation [41, 45]. OR-Parallelism is useful in programs which are heavily non-deterministic as, for example, in search based applications.

Since at least the naive implementation of OR-Parallelism is relatively

straightforward and well understood, the rest of this chapter will concentrate on analyzing the particular problems associated with the implementation of AND-Parallelism. As mentioned before, the independence of the search rule from the computation rule in a Logic Programming system makes it possible to apply the techniques developed for OR-Parallelism implementation in conjunction with those which will be introduced in the next sections (and in the next chapters) for AND-Parallelism.

3.4 AND-Parallelism

AND-Parallelism, in contrast with OR-Parallelism, promises results even for highly deterministic programs. All work done by a collection of AND-Parallel processes is "useful" for finding a particular solution to a query: because *computation rules* are generally exhaustive, it is always necessary to explore *all* paths under an AND node of the search tree. However, AND-Parallelism presents a series of problems which have for some time limited its application to only trivial cases. Most of these problems arise from the fact that goals in the body of a clause which are candidates for AND-Parallel execution often share variables between them and are therefore not independent. A *variable binding conflict* appears if various goals attempt to bind such a shared variable to different values.

3.4.1 All Solutions AND-Parallelism

There is one way of exploiting AND-Parallelism which is based on completely *avoiding* variable binding conflicts. This can be accomplished by *having the goals involved work on* **different solutions** *simultaneously.* Evaluation methods which make use of this technique have been grouped under the name of *all solutions AND-Parallelism.* We will illustrate some of these methods using the `"crew"` example of previous sections. Suppose we have arrived at the point where the goals to solve are

```
:- pilot(Loner), radio_operator(Loner).
```

One possible parallel solution is to apply a *join* algorithm: two processes are started in parallel, one computing *all* the solutions for `pilot(Loner)` and another one computing all the solutions for `radio_operator(Loner)`. After these two sets are computed, their *join* (i.e. the intersection of both sets of solutions) is determined, and it represents the set of solutions for the clause. This mode of operation is called the **join method** or "set at a time" computation [52] and can be useful when computing *all* the solutions for a given query. The main drawbacks of this method are the computational expense of the join operation in practice (unless very specialized hardware is employed), the added complexity introduced in this operation by the presence of *several* variable binding conflicts, and the potentially very large space which may be needed for the storage of intermediate solutions. Conery offers further arguments against this method [20].

A more practical alternative to the join method is the *nested loops* method used by Prolog: for example, returning to the "crew" clause, each solution *found* for `pilot(Loner)` is then *checked* by `radio_operator(Loner)`. The advantage in this method is that the amount of computation can be minimized by correct ordering of goals [85]. There are at least two ways in which all solutions AND-Parallelism can be taken advantage of in the nested loops method:

- As soon as a solution for `pilot` is found it is passed on to `radio_operator` which starts checking it, but at the same time `pilot` continues to look for other solutions. This form of AND-Parallelism is called *pipelining parallelism* by Tamura and Kaneda [74]. In their scheme the "presearch" for solutions done by `pilot` would only proceed up to a number of them called the *buffer size*. A buffer size of 0 results in sequential Prolog execution.

- An alternative to the scheme above is to start a new "checking" process evaluating `radio_operator` for each new solution computed by the process evaluating `pilot`. This approach clearly generates more parallelism than the one above.

These approaches suffer from some of the same drawbacks as OR-Parallelism: although the search space can be bound in them (for example by using the limited buffer size solution of Tamura and Kaneda), they still rely on the presence of non-determinism (i.e. multiple solutions) in the problem in order to attain parallelism.

3.4.2 Variable Binding Conflicts in AND-Parallelism

In contrast with the approaches described in the previous section, there are methods which can take advantage of AND-Parallelism even for highly determinate programs. This can be accomplished by *having parallel processes work on the* **same solution**. However, we have already shown how "brute force" [20] exploitation of this type of AND-Parallelism (i.e. the automatic scheduling of a process for every goal in the body of a clause) can potentially lead to binding conflicts if the goals involved have variables in common. The appearance of *Variable Binding Conflicts* during AND-Parallel execution was already apparent in the example in section 3.1.2. The intuitive idea there was that these conflicts appeared only in clauses whose goals shared variables in the program. However, in practice these conflicts can occur even in cases where the goals appear not to share variables at all. Consider the following simplified version of the "crew" example

```
crew(X,Y):- pilot(X), radio_operator(Y).
```

and the query

```
?:- crew(ian, sabina).
```

It is obvious then that checking if `"pilot(ian)"` and if `"radio_operator(sabina)"` can be done in parallel (AND-Parallelism). Suppose, however, that the question is whether there is anybody who can fly a plane without need for other crew members, i.e. if there is somebody who is a pilot and can also operate a radio. This question would be stated as

```
?:- crew(Loner,Loner).
```

During the unification of this query with the head of the "crew" clause, the substitutions would be Loner/X and Loner/Y and the apparently **independent** variables X and Y in the clause above would be coerced to be the same (and bound to Loner). Therefore, after unification, the resulting new query would be

```
:- pilot(Loner), radio_operator(Loner).
```

In this case, it is not possible to go ahead and evaluate these two goals in parallel because of the potential for conflicting instantiations of Loner. These goals have been determined to be **dependent**, but note that this determination was only possible **at run time**, i.e. once it is known that the variables X and Y *share* as a result of unification with this particular form of the query.

Fortunately, the inverse case of the one shown above is also often true: in some instances, even though variables may appear to be shared by some goals in the body of a clause, execution can actually proceed in parallel. Consider the following clause

```
child(X,Y,Z):- father(Y,X), mother(Z,X).
```

where father and mother clearly share the variable X. For and the query "who is a child of Peter and Mary?":

```
?:- child(C, peter, mary).
```

the resulting goals

```
:- father(peter,C), mother(mary,C).
```

offer potential for a variable binding conflict for C. However, consider the problem of finding the parents of Peter:

```
?:- child(peter, F, M).
```

which results in the goals

```
:- father(F,peter), mother(M,peter).
```

Finding the answer for these two goals can now be done in parallel, because the variable shared between the goals is "ground" during execution, i.e. it has been instantiated to a term containing no free variables before the goals are called.

As a result of the considerations presented in the previous paragraphs there appear to be two main issues involved in handling AND-Parallel execution of goals working on the same solution:

- *Detecting Variable Binding Conflicts:* identifying the cases where these conflicts actually occur. This detection can be difficult in practice and a potential source of overhead, since, as shown in the previous paragraphs, at least some of the detection has to be done at run-time.

- *Dealing with Variable Binding Conflicts:* once a conflict is detected, deciding the course of action to be taken in order to proceed with execution either sequentially or, if at all possible, in parallel.

There are many possible approaches which have been proposed for detecting and dealing with variable binding conflicts. In the following sections we will review some of the techniques currently used pointing out their relative advantages.

3.4.2.1. Dealing with Variable Binding Conflicts

The three basic methods of dealing with variable binding conflicts were pointed out in the example in section 3.1.2:

- **Goal Suspension:** All goals are started in parallel, but goals which are consumers of variables which have not been fully instantiated yet wait (*suspend*) until the instantiation is complete. The main problem in this approach is the complicated run-time system involved, which continuously has to keep track of the instantiation state of variables.

- **Stream AND-Parallelism:** for each shared variable, one goal is determined as the *producer* of the variable, and the others as consumers. All goals in the body are run in parallel and the value of the variable is *incrementally* passed from the producer to the consumers. Stream AND-Parallelism can take advantage of most of the potential AND-Parallelism

present in the clause. The main drawbacks are the low level of granularity (which may make it difficult to implement in an efficient way) and the fact that it is very difficult to support in the presence of non-determinism.

- **Goal Independence:** goals which are determined to be independent (i.e. there are no possible variable binding conflicts) are run in parallel, otherwise, they are run sequentially. The problem with this approach is the overhead involved in the determination of this independence. Simple techniques which do not need run-time support often fail to detect potential parallelism. Run-time based approaches often incur in excessive overhead.

3.4.2.2. Detecting Variable Binding Conflicts

User Annotation: The simplest approach for detecting conflicts is, of course, to have the programmer determine goals which are guaranteed to be independent. There are two basic techniques for expressing this information:

- *Goal Annotation:* programming languages which support this method provide the programmer with an *extended control language* which makes it possible to *annotate* sets of goals as candidates for AND-Parallel execution (IC-Prolog [14], Delta Prolog [57]). For example, in Delta Prolog parallel composition of goals is annotated by linking those goals with the connective '/', while ',' is still used for expressing sequentiality.

- *Variable Annotation:* A refinement of the approach described above which directly points out the conflicts is to mark the potentially shared variables in the clause so that goals involving these variables will wait until they are fully instantiated (this is the type of annotation usually associated with goal suspension [63]). In systems which can support simultaneous execution even in the presence of variable conflicts (*stream AND-Parallel models*) the same mechanism is used not only for marking potential conflicts, but also for defining the direction of stream communication, i.e. which goal is the producer and which goals are the consumers for a given variable (Concurrent Prolog [72]). A similar approach which can be used in these systems is to declare *modes* for the variables in the heads of the clauses: by defining arguments in the head of a clause as input-only or output-only the direction of the streams can be determined. This method is used in PARLOG [13].

Automatic Detection: Solutions such as the above put the burden of

detecting variable binding conflicts in the hands of the programmer. This can be an acceptable solution in many applications, but it clearly defeats the objective of hiding as much as possible control-related issues from the user. Therefore, other solutions have been proposed which attempt to detect binding conflicts without variable annotations and with minimal (or no) information from the user. These approaches differ mainly in the ratio of the amount of work done at compile-time to that done at run-time:

- *Run-time Detection:* Of course variable conflicts can be easily detected at run-time. However, the amount of overhead incurred in doing so often makes the approach impracticable [20].

- *Compile-time Analysis:* Several techniques have been proposed which try to perform a compile-time analysis of the data dependencies in the program in order to determine variable independence. This analysis is specially complex in Logic Programs because of the bidirectionality of the Logic Variable. However, input and output modes can in some cases be determined by such an analysis using the known modes of built-in predicates and/or some user "hints" (such as, for example, the types of queries which can be expected [24] [51] [48] [49]). A similar analysis can be used to determine goals which are guaranteed to be independent (i.e. no variable binding conflicts will be encountered) so that no checks are needed at run-time [9]. The main problem with such a system is that only one possible mode of operation (one input-output pattern) is allowed for parallel operation.

- *Combined Approach:* An approach combining compile-time and run-time techniques can analyze *several* possible input-output patterns and define different sets of goals as independent as a function of the actual input-output pattern which occurs at run-time. The run-time detection of the pattern represents a small overhead compared with performing a complete data dependency analysis for each invocation of a clause (as in run-time detection systems [20]). One such method, based on analyzing the state of instantiation of a set of variables is used in DeGroot's *restricted AND-Parallelism* [25].

3.4.3 Proposed Systems Supporting AND-Parallelism

In the previous section we presented some of the techniques which can be applied while detecting and dealing with variable binding conflicts in AND-Parallelism. In this section we will review some proposed systems which make use of one or more of those techniques. Although most of these systems were already mentioned in previous sections, a more detailed description will now be given, pointing out their relative advantages.

3.4.3.1. Committed Choice Systems

As described previously, one solution to the problem of dealing with variable binding conflicts is stream AND-Parallelism: one goal is determined as the *producer* of each shared variable, and the others as *consumers*. These goals then all run in parallel and the value of the variable is *incrementally* passed from the producer to the consumers. This mode of communication between goals is very useful in that it allows the description of systems of communicating processes. One disadvantage, though, is that the low level of granularity involved in stream parallelism seems to make it difficult to implement in an efficient way. This problem could of course be solved in a specialized architecture. The main drawback in stream AND-Parallelism, however, is that it is very difficult to implement in the presence of non-determinism. Therefore, recently proposed systems which exploit this type of parallelism *give up true non-deterministic search* by implementing "committed-choice" (i.e. "don't care") non-determinism: once a path in the execution tree is chosen, no other paths will be explored. These systems are somewhat closer to functional languages in the sense that clauses behave as functions, providing only *one* solution to a given query. PARLOG [30] [13], Concurrent Prolog [72], and GHC [81] are examples of "committed choice" languages.

However, "don't-know" nondeterminism is regarded as one of the most

interesting features of Logic Programming. As we have seen before, there are methods of dealing with variable binding conflicts other than stream AND-Parallelism which naturally support both AND-Parallelism and "don't know" non determinism. Some previously proposed approaches which make use of such methods will be presented in the next sections.

3.4.3.2. Conery's AND/OR process model

Conery describes an "AND/OR process" execution model which is a distributed, message-based scheme capable of handling full "don't-know" non-deterministic parallel execution of Logic Programs. The main types of parallelism supported are OR- and AND-Parallelism. Conery shows how OR-Parallelism can be supported in a straightforward manner and also how in general AND-Parallelism is much more difficult to support because of the problems introduced by the sharing of variables in literals within a clause body. Conery's AND process model offers a solution for this problem which can extract most of the parallelism available for a given collection of AND-Parallel goals. He introduces a series of run-time algorithms which can determine goal ordering, producer selection and the correct points for parallel backtracking. This information is represented in the form of data dependency graphs. The algorithms effectively extract the available degree of parallelism present in each particular clause invocation and offer a complete and powerful solution to the management of the AND sections of the execution tree.

The main drawback in Conery's scheme is the enormous amount of run-time support necessary to implement its operation: all his algorithms produce results which depend on the particular instantiations of the variables involved, so the dependency graphs have to be recomputed for each clause invocation and upon backtracking. This results in an unacceptable amount of overhead that probably renders an otherwise very attractive theoretical model rather impractical.

One of the merits of this work has been to provide the first complete solution to the problem of correctly handling AND-Parallelism, performing an interesting analysis of the problems involved. Other related models have since been proposed [43] [62] [42] and Conery's model has thus proven very useful in setting the grounds for other schemes. Many of these schemes are based on the same ideas but try to overcome the drawbacks of Conery's model by extracting as much as possible of the information required for correct execution of AND-Parallel clauses during compilation. Since these approaches then require little or no run-time support, faster execution can be truly achieved. Two of these schemes are presented in the following paragraphs.

3.4.3.3. Static Data Dependency Analysis

The idea behind Chang's Static Data Dependency Analysis (SDDA) [9] is to derive data dependency graphs at compile time from a small amount of additional information supplied by the programmer: the "activation mode" of the query. This means that the programmer has to supply the SDDA analyzer with information on which particular queries are going to be presented to the program (i.e. which procedures are going to be called) and which of the arguments in the call are going to be ground, independent or dependent. The output of the analysis is a graph which determines which goals in the clause bodies are independent and can be thus run in parallel, in which order the non-independent goals have to be run, and a compatible scheme for semi-intelligent backtracking.

The main advantage of this approach is that no run-time support is needed for variable binding conflict detection, and the fact that working algorithms are available in order to generate the above mentioned graphs. One drawback is that, since only one type of query is allowed for each procedure, other queries which do not adhere to the declared activation mode will not be executed in parallel or make use of the semi-intelligent backtracking at all. Furthermore, since the approach is

necessarily based on a worst-case analysis (since so little about the bindings of variables is known at compile time) it has the danger of often missing some of the parallelism available even for the particular type of query analyzed. Finally, the semi-intelligent backtracking scheme is rather complicated, requiring a fair amount of run-time support.

3.4.3.4. Restricted AND-Parallelism

The approach taken by DeGroot in his *Restricted And-Parallelism* (**RAP**) scheme [25] is to choose a compromise solution between complete run-time (Conery) and complete compile-time (Chang) determination of data dependencies between goals in the body of a clause. Instead of determining only one worst case data dependency graph at compile time, several graphs are generated for each clause, each one of them valid for a particular activation mode of the clause. These graphs are then combined into a single *Conditional Graph Expression* (**CGE**). The run-time system, while executing the **CGE**, will choose one of the different graphs for each activation of the clause depending on the results of a set of simple run-time *checks* (included at compile-time in the **CGE**) which determine which variables in the clause are independent. The graph selected will be one that starts execution in parallel of the goals involving only those variables.

The set of possible expressions is defined as:

(1) **G** An arbitrary goal.

(2) **(SEQ E1 ... EN)**
Expressions **E1** to **EN** are to be run sequentially.

(3) **(PAR E1 ... EN)**
Expressions **E1** to **EN** are to be run in parallel.

(4) **(IPAR (X1 ... XN) E1 ... EN)**

Expressions **E1** to **EN** are to be run in parallel if **(X1 ... XN)** are *independent*, otherwise they are to be run sequentially.

(5) **(GPAR (X1 ... XN) E1 ... EN)**
Expressions **E1** to **EN** are to be run in parallel if **(X1 ... XN)** are *ground*, otherwise they are to be run sequentially.

(6) **(IF E1 E2 E3)**
Chooses between evaluation of **E2** or **E3** depending on the result of evaluating the boolean expression **E1**.

An example will clarify the use of these expressions further. Recall the "child" example:

```
child(X,Y,Z):- father(Y,X), mother(Z,X).
```

In Restricted AND-Parallelism the compiler would analyze this clause and come to similar conclusions to those pointed out when the example was introduced in previous sections: for example, it can decide that "`father(Y,X)`" and "`mother(Z,X)`" cannot in general run in parallel, but that it is possible to execute them in parallel if the clause happens to be called with all arguments (`X`, `Y`, and `Z`) being ground: This information can be encoded in a *Conditional Graph Expression*:

```
(GPAR(X Y Z)  father(Y,X)  mother(Z,X) )
```

The meaning of the expression above is

> If `X`, `Y`, and `Z` are ground, `father(Y,X)` and `mother(Z,X)` can run in parallel, else, they are to be run sequentially.

Thus, the expression above can generate (depending on the results of the conditions) two execution graphs at run-time: a sequential and a parallel one. Nesting of **CGE**s and conditions can generate more complicated execution graphs.

DeGroot's expressions have however some limitations. For example, the

compiler could also have observed that "father(Y,X)" and "mother(Z,X)" can also run in parallel if the clause is called with the first argument (X) being "ground" (i.e. fully instantiated -it contains no variables) and the other two (Y and Z) being "independent" (i.e. X and Y do not "share"). This information is difficult to encode with the above expressions. DeGroot also points out other such limitations [25].

Clearly, the generation of the CGE in the "child" clause (i.e. determining that there are basically two interesting cases which can appear at run-time for this clause) can be done at compile-time, but the actual checking in order to find out in which particular case the clause is being executed can only be done at run-time. Conery's approach would perform all these operations at run-time, while Chang's would do a similar analysis at compile-time but it would have to select the worst of all possible cases for lack of run-time checks.

DeGroot's Restricted AND-Parallelism scheme offers advantages over both of the approaches described before: the run-time system is obviously much simpler than that needed in Conery's AND-process and it is simpler and offers potential for exploiting parallelism in more cases than Chang's worst case analysis. The main disadvantages present in this description of Restricted AND-Parallelism are the intrinsic limitations of the expressions proposed and the fact that the backward execution behavior of these expressions (i.e. how to handle failure during parallel execution) is not specified. Also, there is no indication as to what algorithm or heuristics should be used to generate such expressions.

3.5 Chapter Summary: A Proposed Approach to Parallel Logic Programming Implementation

In the previous sections the different sources of parallelism present in Logic Programs were introduced and some of the problems which arise in the exploitation of these sources were reviewed. OR-Parallelism was shown to present a series of drawbacks in practice, such as only being able to extract useful parallelism in non-deterministic problems. Also, it was pointed out how it can require excessive amounts of storage and/or copying time and present a combinatorial explosion in the size of the search space to be explored, and in the number of processes generated. Furthermore, much of the work done by a collection of OR-Processes is often not considered "useful work" for arriving at a particular solution.

AND-Parallelism, on the other hand, was shown to be especially interesting because it can provide performance improvements even in the absence of non-determinism in the problem and because, in general, all work done by a collection of AND-Parallel processes is "useful" for finding a particular solution to a query. However, AND-Parallelism was also shown to present some problems, such as detecting and dealing with variable binding conflicts and its incompatibility with non-determinism in some approaches. Nevertheless, if these conflicts can be dealt with without excessive overhead, AND-Parallelism appears to offer more useful parallelism and with a more efficient utilization of the available resources.

Following the above considerations it is concluded that *a higher emphasis in parallel logic program implementation should be put on supporting AND-Parallelism*, although the problems associated with its implementation need to be addressed. Of course, an ideal system should be able to support both OR- and AND-Parallelism (and perhaps the other types of parallelism as well). Since the issues

involved in the implementation of OR-Parallelism are generally better understood that those associated with AND-Parallelism, the following chapters will be devoted to the study of AND-Parallelism in the conviction that the techniques developed will also be useful in a system incorporating both of the basic sources of parallelism.

Regarding the way in which the inherent problems in AND-Parallelism implementation are to be treated, stream AND-Parallelism was shown to offer an interesting potential for deterministic execution but it appeared as very difficult to implement in the presence of non-determinism. "Committed-choice" systems were shown to support stream AND-Parallelism by giving up true non-deterministic search and implementing "don't care" non-determinism. However, it was also mentioned how "don't-know" nondeterminism is regarded as one of the most interesting features of Logic Programming and how there is another way of dealing with variable binding conflicts which naturally supports both AND-Parallelism and "don't know" non determinism: restricting AND-parallel execution to sets of goals which are *determined to be independent at run-time.*

In the next chapters we will address the design of an *efficient* execution model for the parallel implementation of Logic Programs, capable of supporting AND-Parallelism in the presence of "don't-know" non-determinism. The emphasis will not be on a particular language, but rather on developing techniques which can be applied to a variety of languages which support this type of non-determinism, and also to the "don't know" subsystem of committed choice languages. Although previous similar approaches have resulted in excessive run-time overhead or limited parallelism we will show in the next chapters how AND-Parallelism supporting full non-determinism can in fact be implemented very efficiently by combining a generalized version of Restricted AND-Parallelism, *Goal Independence Parallelism*, with some of the

implementation techniques of current high performance sequential systems. Also, limitations of, and areas missing in, previous descriptions of these models will be addressed, such as providing a simpler and more powerful set of Conditional Graph Expressions and offering complete (forward and backward) procedural semantics for logic clauses annotated with these expressions.

Chapter 4

A High-Level Execution Model for AND-Parallelism: Procedural Semantics

In this and the next chapters the design of an *efficient* execution model for the parallel implementation of Logic Programs capable of supporting *AND-Parallelism* in the presence of *"don't-know" non-determinism* will be addressed. The organization of the chapter is as follows: first, "goal independence" models of AND-Parallelism will be reviewed and a generalized version of Restricted AND-Parallelism (**RAP**) presented as a typical representative of this class. Areas missing in previous descriptions of these models will be completed, such as providing complete (forward and backward) procedural semantics for Horn clauses which have *Conditional Graph Expressions* embedded within them, and a more powerful definition and syntax will be given for these expressions. Some consideration will also be given to the necessary conditions which have to be met by goal independence annotations such as **CGE**s. Finally, a programmer's view of such a system will be presented.

4.1 A General Model for AND-Parallelism: Goal Independence

It was mentioned in the previous chapter how "brute force" exploitation of AND-Parallelism (i.e. the automatic scheduling of a process for every goal in the body of a clause) leads to binding conflicts if the goals involved have variables in common. However, if these goals can be determined to be independent at run-time, execution can still continue in parallel. This can be termed **Goal Independence AND-Parallelism**, i.e. AND-Parallel execution in which source level annotations and/or run-time mechanisms are geared towards determining a set of goals as being independent at some point in the execution of a particular clause.

It was also shown how in some logic programming languages *goal independence* is annotated by the programmer explicitly in the source program. This is the case, for example, in Delta-Prolog [57], a partial implementation of distributed logic [50]. In this model, parallel composition of goals is annotated by linking those goals with the connective '/', while ',' is still used for expressing sequentiality. Thus, the following clause

```
f(X, Y, Z) :- ( a(foo1, X) /  b(foo2, Y) ), c(X, Y, Z) .
```

basically expresses that goals **a** and **b** are mutually independent, i.e. that in the resolution of **f**, **a** can be executed independently of **b** (i.e. in parallel, and in any order), but **c** has to wait for both of them to succeed. Thus, the annotation really expresses an execution graph. Note, however, that an implicit assumption was made in the clause above that a query of the form "`?:- f(W, W, Z).`" is not possible, that is, that "`X`" and "`Y`" will never share. Thus the execution graph implied by the annotation in the clause above is really only valid for a particular type of query. Other similar types of annotations are present in many other languages which provide some form of *goal independence* AND-Parallelism (IC-Prolog [14], PRISM [37]). It was also mentioned how in some other languages the determination of goal

independence is done by the compiler, often guided by some information provided by the user on the type of queries that are most likely to be presented to the system [9]. For example, in Chang's approach, if the user declares that the most likely query is of the form

```
?:- f( ground_term, ground_term, free_variable).
```

an execution graph equivalent to the Delta-Prolog annotated clause above could be generated automatically by the compiler.

Because we are more interested in developing an execution model for AND-Parallelism than in the study of any particular Logic Programming language (hopefully the execution model will be applicable to a *variety* of languages) we will not be concerned at this point with the *origin* of the annotations which determine goal independence, although a starting point for the automatic generation of such annotations will be given at the end of the chapter. Instead, we will concentrate on determining which *types of annotations* (user- or compiler-generated) offer maximum potential for parallelism with minimum run-time cost and on dealing with *how execution proceeds* once a set of goals has been determined as being (variable-wise) independent (i.e. after determining that they can be run in parallel with no conflicts), in particular on how "don't know" non-determinism can still be efficiently supported in such an environment.

Consequently, rather than analyzing the language at the source-level, we will focus on an *intermediate code level* useful for a variety of programming languages, and we will pursue development of an efficient execution model for it. This level, which will be discussed in the next section, can be best described as *horn clauses augmented with predicate-level conditional control expressions.* Such control expressions can, for example, be generated when a static analysis uncovers parallel

execution potential. Alternatively, the source language could provide the user with the syntactic tools to explicitly trigger their generation.

Concerning the character of these expressions, it has already been pointed out how in logic programs, the same clause can be used in various ways, depending on the run-time polarity (instantiation state) of interceding variables. Ideally, these expressions should be capable of dealing with the different cases involved, with a minimum of run-time overhead. As already mentioned in the previous chapter, *Restricted AND-Parallelism* (**RAP**) [25] is a technique which provides this capability by making it possible to choose at run-time between parallel and sequential execution (i.e. to generate one of several possible execution graphs) based on variable dependency checks. Such run-time determinations are embodied in what has been referred to as *Conditional Graph Expressions* (**CGE**'s). In the next section we will present a generalized version of such a computation model which subsumes DeGroot's original definition of **RAP** and **CGE**'s. It will be the forward and backward execution behavior of this generalized model that we will study in the subsequent sections.

4.1.1 Conditional Graph Expressions

As explained above, **CGE**'s can be used for reducing run-time data dependency analysis overhead for AND-Parallel logic programming systems to a number of simple checks. Herein, a **CGE** is (informally) defined as a series of conditions followed by a conjunction of goals, i.e.:

```
( <CONDITIONS> | goal1 & goal2 & ... & goalN )
```

where "<CONDITIONS>" represents *any number of conjunctions or disjunctions of checks on a* **<variable_list>**. A **<variable_list>** is a collection of variable names which have their first occurrence before (i.e. "to the left of", in Prolog) the

<CONDITIONS> field of the current graph expression[24] . In this definition **CGE**'s can appear in the body of a clause in any place a conventional goal may be placed. Therefore they can also appear in a goal position *inside* a **CGE** (nested **CGE**'s). Types of checks which can appear inside <CONDITIONS> are:

- **ground(<variable_list>)**: evaluates to ***true*** if and only if all variables in **<variable_list>** are ground, i.e. they are instantiated to a term with no uninstantiated variables.

- **indep(<variable_list>)**: We associate with each variable its "set of contained variables" (**SCV**), defined as follows: If the variable is instantiated to a fully ground term, the **SCV** is ***empty***. If the variable is uninstantiated, the **SCV** is the singleton containing the variable itself. If the variable is instantiated to a term, and some of its arguments are variables, the **SCV** is recursively defined as the union of the **SCV**'s for each of those variables. The **indep(<variable_list>)** check succeeds if and only if the intersection of all the **SCV**'s associated with each variable in **< variable_list >** is ***empty***[25] .

- The logical values **true** and **false**.

4.1.2 Forward Execution

Since each of the checks inside **<CONDITIONS>** will evaluate to **true** or **false**, **<CONDITIONS>**, being constructed as conjunctions and/or disjunctions of these checks, will also eventually evaluate to **true** or **false**. The forward semantics of **CGE**'s dictates that:

> *if <CONDITIONS> evaluates to true, then all expressions inside the **CGE can** execute in parallel. Otherwise, they **must** be executed sequentially and in the order in which they appear within the expression.*

---

[24] i.e. only those variables in the head or in goals to the left of the current **CGE** (including those in a **CGE** the current expression may be nested in) can be checked.

[25] Much more economical independence algorithms (such as DeGroot's [25]) can be used in practice, as long as they are conservative, i.e. they never declare a set of dependent variables as independent (although they may "give up" and declare some variables as dependent rather than traversing very complex terms).

A **CGE** whose **<CONDITIONS>** have evaluated to **true** is called a **Parallel Call.** An example will clarify this further. Suppose we have the following clause:

```
f(X,Y) :- g(X,Y), h(X), k(Y).
```

In general, the three goals in the body of **f** (**g**, **h** and **k**) cannot run in parallel because they have variables in common. Nevertheless, if both **X** and **Y** are ground when **f** is called, all goals can then run in parallel. This fact can be expressed by using the following **CGE**:

```
f(X,Y) :- ( ground(X,Y) | g(X,Y) & h(X) & k(Y) )
```

According to the forward execution semantics above, this means that **X** and **Y** should be checked and, if they are both ground, then **g**, **h**, and **k** can be executed in parallel and execution will proceed to the right of the expression only after all goals inside succeed. Note that this also means that if **X** and **Y** are ground but for some reason (for example, lack of free processors) **g**, **h**, and **k** are executed sequentially, this can be done in any order. Otherwise, if **X** and **Y** are not both ground, **g**, **h**, and **k** will run sequentially and in the order in which they appear inside the **CGE**. Selection between one mode of execution and the other is done by a simple run-time check. Of course, the expression above only takes care of a rather trivial case.

A more interesting execution behavior can be extracted from the following expression:

```
f(X,Y) :- ( ground(X,Y) | g(X,Y) & ( indep(X,Y) | h(X) & k(Y) ) ).
```

Now, if X and Y are not ground upon entry to the graph expression, **g** will be executed first. As soon as **g** succeeds, `indep(X,Y)` is checked in the hope that X and Y will be independent (either because one of them was ground by **g** or because they are still uninstantiated and do not "share" --as they would if **g** had matched against

" :-g(W,W) ."). If they are still independent then **h** and **k** can run in parallel. Note that if **X** and **Y** are ground upon entry of **f** then *all* goals will run in parallel as in the previous expression.

Sometimes it is necessary to express the fact that a number of goals can run in parallel, independently of any other consideration (perhaps because the programmer knows how a procedure is going to be used). This can be easily accomplished by writing **true** in place of **<conditions>** or eliminating the **<conditions>** field altogether. Thus, in the following expressions, **g**, **h**, and **k** can always run in parallel:

```
f(X,Y) :- ( true | g(X) & h(Y) & k(Z) ).
f(X,Y) :- ( g(X) & h(Y) & k(Z) ).
```

This also illustrates how **CGE**'s are a superset of other control annotation schemes (for example, the parallel connective of Delta-Prolog "/") [57].

Note how in this definition Conditional Graph Expressions do not need to be considered as independent constructs from the original clauses (as implied by DeGroot [25]). In the examples above they can be viewed as "compiler generated" annotations to the (unannotated) source program or alternatively as user directives embedded in it. We will come back to the issue of considering **CGE**s as annotations embedded within logic clauses in the discussion of the programmer's view of the Restricted AND-Parallel (RAP) system at the end of the chapter.

4.1.3 Backward Execution

We refer to backward execution as the series of actions that follow **failure**. Failure occurs during unification of a given goal with the head of a clause, if this unification does not succeed. Is was shown in Chapter 2 how, for example, in a system supporting depth first search with backtracking (such as Prolog), these actions simply comprise returning to the most recent point at which alternatives were still unexplored (i.e. the last time a clause was entered which had other alternative clauses which could also have unified with the current goal) and continuing execution with the next alternative. If full ("pure") OR-Parallelism is supported (perhaps in addition to AND-Parallelism) *backtracking* is not needed; a set of "solutions" is maintained instead for each goal invocation. Failure implies simply abandoning the path being followed, because the alternatives are already being explored by other processes.

While the relative simplicity of such an approach and the additional source of parallelism make it attractive in principle, keeping multiple solutions around simultaneously obviously tends to complicate data storage management and use up excessive amounts of this storage. Moreover, as pointed out in the previous chapter, the additional parallelism often leads to a combinatorial explosion of the search space. Therefore, even systems which rely only on OR-Parallelism limit its occurrence to only certain branching points previously annotated by the user or as a result of detailed compiler analysis. Therefore, backtracking still has to be supported even in OR-Parallel systems in order to deal with non-parallel OR-nodes. This is even more so if both AND- and OR-Parallelism are supported: since AND-Parallelism is generally given a higher priority than OR-Parallelism, as soon as spare resources start being scarce in the system, backtracking will have to be used in lieu of OR-Parallelism in order to reserve those resources for AND-Parallel execution. Also, if the system is limited in memory, backtracking offers a solution which is more memory efficient than

OR-Parallelism. For the reasons above, and since the treatment of failure in pure OR-Parallelism simply comprises the set of simple actions needed in order to cease execution of a search path, this section will study backward execution algorithms for Horn clauses annotated with **CGE**'s in parallel systems for cases in which failure implies *backtracking*.

The simple treatment of backward execution in sequential systems (i.e. simply returning to the last choice point and restarting execution with the next alternative) is not directly applicable any more if some of the goals in the body of a clause have been executed in parallel: since execution of these goals was concurrent, there is no chronological notion of "most recent" to apply to the different choice points available. Although several sophisticated approaches have been proposed in order to solve this problem [20] [9] [56] [7] they are either not applicable to the semantics of **CGE**'s (and other Goal Independence models) or they involve too much bookkeeping overhead at run-time. In this section we will analyze the different cases involved in the backtracking of **CGE**'s and we will propose a general backtracking algorithm that will handle these cases efficiently, while taking advantage in some cases of goal independence in order to achieve a limited for of intelligent backtracking [59] [60]. This will be referred to as "restricted" intelligent backtracking.

4.1.3.1. Backtracking Cases

Throughout this analysis we will consider the following annotated clause[26]:

`f(..):- a(..), b(..),` (<**conditions**>| c(..) & d(..) & e(..)), `g(..), h(..).`

---

[26] Although the discussions in this chapter will not directly address nested **CGE**'s, the algorithms shown are also applicable in such cases when applied recursively. Alternatively, a clause with nested **CGE**s can be trivially reduced to a set of clauses with non-nested **CGE**s by substituting each nested **CGE** in the original clause by a call to a "dummy" goal whose corresponding clause simply embodies the nested **CGE**.

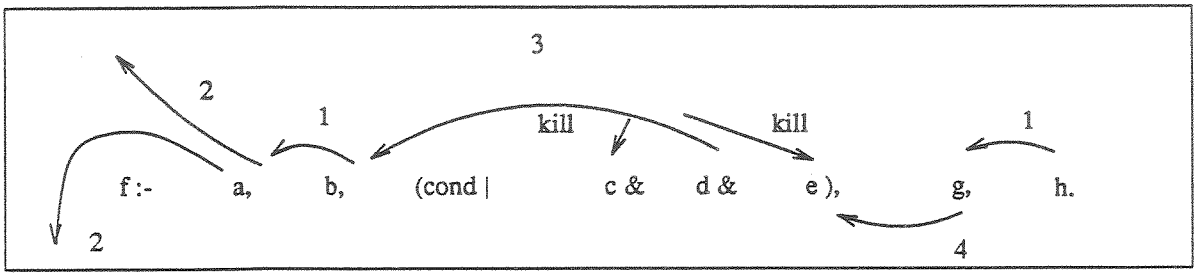

Figure 4-1: Backtracking cases for a CGE

In the trivial case when **<conditions>** is evaluated to false, execution defaults to sequential, and normal (Prolog) backtracking semantics can obviously be applied. We will therefore shift our attention to the cases where **<conditions>** evaluates to **true**. We illustrate in figure 4-1 the different backtracking situations through back arrows annotated by case numbers, where the cases are the subject of the following text.

Conventional Backtracking:

- **Case 1-** This is the trivial case in which backtracking still remains the same as for sequential execution. For example, if **b** fails and **a** still has alternatives, or if **h** fails and **g** still has alternatives.

- **Case 2-** This is also a trivial case: if **a** fails, the next alternative of **f** will be executed next. If there are no more alternatives for **f**, then **f** will fail in its parent and we recursively deal with the failure at that level.

Conjunctive failure; "inside" backtracking:

- **Case 3-** This is the case if **c**, **d**, or **e** fail while the body of the CGE is being executed the first time through (i.e. we are still "**inside**" the CGE).

  Suppose **d** fails. Since we are running in parallel, we know that **<conditions>** evaluated to true. This means that **c**, **d**, and **e** do not share any uninstantiated variables. Thus, the variable binding that caused

the failure of **d** could not have been generated by **c** or **e**. Therefore it would be useless to ask **c** and/or **e** for alternatives and it is safe to **kill** the processes running **c**, **d**, and **e**, and to backtrack to the most recent choice point before the **CGE** (for example, **b** here). In this way, limited (restricted) intelligent backtracking takes place inside the **CGE** with only the overhead of remembering that we are **"inside"** the **CGE** when failure occurs.

"Outside" backtracking: ("Point method")

- **Case 4-** This is the most interesting case: we have already finished executing all goals inside the **CGE** -we are **"outside"** the **CGE**- and we fail, having to backtrack into the expression. This is the case if **g** fails.

  First, since this information will prove very useful, we will assume that processes not only report eventual goal resolution success, but also whether unexplored alternatives still remain for this goal. It will be shown how such information can be used in our context to simply extend the conventional backtracking algorithm to one that deals with **CGE**'s:

  - If **g** fails and none of the **CGE** goals has unexplored alternatives, we will backtrack to **b** just as we would in the sequential execution model.

  - If **g** fails and one or more **CGE** goals still has unexplored alternatives, our object will be to establish a methodology whereby all the combinations of those alternatives will have a chance to be explored, if needed, before we give up on the whole **CGE** and backtrack to alternatives prior to it. The methodology chosen is one that will generate those alternatives *in the same order as that produced by naive sequential backtracking.* The idea is then to reinvoke the process which corresponds to the first goal with alternatives found when scanning the **CGE** *in reverse order* (i.e. reinvoking the "rightmost" goal with alternatives). All processes corresponding to goals "to the right" of this one will be **"unwound"** (i.e. the bindings they created undone)[27]. The reinvoked process will then, in turn, report either **success** (with or without pending alternatives) or **failure**.

---

[27] Note that these processes are not actually running but it is advantageous to deallocate the storage used for computing their last alternative at this point. These issues are discussed in more detail in the following chapter.

- If **failure** is reported, we simply perform the next invocation in the order described above. Of course when a **failure** is reported by the leftmost goal with alternatives in the **CGE**, we give up on the whole expression and backtrack as in **Case 1** above.

- If **success** is reported (i.e. a **point** of success is found in the **CGE**) then we shift into forward AND-Parallel execution mode and *trigger the parallel evaluation of all the goals, if any exist, to the right of the succeeding one in the* **CGE**.

Note how the approach described above extends the "most recent choice point" backtracking model to a parallel execution model, preserving the generation of all elements of the cross product (i.e. all "tuples") and offering parallel forward execution after backtracking. Also, goal ordering information provided by the user or by the compiler is preserved, and used in tuple generation.

4.1.3.2. Determinate Execution

Alternatively, sometimes we might not be interested in generating all possible tuples for a conjunction of independent goals. Instead we might be interested in generating only one and "committing" to it. This can be easily annotated by including a "cut"[28] after the **CGE**. In the following clause

`f(..):- a(..), b(..),` **(<conditions>| c(..) & d(..) & e(..))**`, !, g(..), h(..).`

the *cut* operator forces the system to commit to the first solution obtained from the invocation of the **CGE** (and since the call which invoked **f**). This case is very interesting because of the potential for efficiency at the implementation level. In the following paragraphs a variation of the backward semantics previously proposed is presented which can be used to take advantage of this potential for efficiency in determinate execution.

---

[28] Note that if only backtracking and AND-Parallelism are supported in the system (i.e. (pure) OR-Parallelism is not used) the semantics of the "cut" operator can remain the same as in sequential systems. If (pure) OR-Parallelism is supported then a "commit" operator has to be used in place of the "cut".

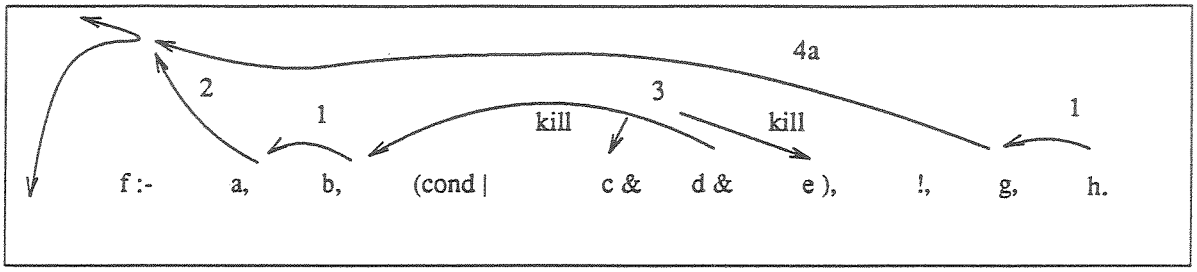

Figure 4-2: Backtracking cases for a CGE: Determinate Execution-(a)

Again, in the trivial case when **<conditions>** is evaluated to false, execution defaults to sequential, and normal (Prolog) backtracking semantics can obviously be applied. We will therefore shift our attention once more to the cases where **<conditions>** evaluates to **true**. The different backtracking situations are now illustrated in figure 4-2, and the corresponding actions are:

Conventional Backtracking:

- **Case 1 and Case 2** - Similar to the non-determinate case.

Conjunctive failure; "inside" backtracking:

- **Case 3-** This is the case if **c**, **d**, or **e** fail while the body of the **CGE** is being executed the first (and only!) time through (i.e. we are still **"inside"** the **CGE**). Again it would be useless to ask any of the other goals for alternatives and it is safe to **kill** the processes running **c**, **d**, and **e**, and to backtrack to the most recent choice point before the **CGE** (for example, **b** here): (restricted) intelligent backtracking.

"Outside" backtracking:

- **Case 4a-** Execution differs now substantially from the non-determinate case: all goals inside the **CGE** have finished executing (past the *cut*) -we are **"outside"** the **CGE**- and we fail. Suppose again that **g** fails. Now, execution has to return to a point *before* the invocation of **f** and no

backtracking needs to be done inside the **CGE**. The only action required is to undo the bindings done by the goals in the **CGE**.

Note that much of the information which needed recording in the non-determinate case (for example, whether processes had pending alternatives or not) is not needed for determinate execution. This will lead to great economy in the implementation. Also note that *all* backtracking as a result of the failure of a goal inside the **CGE** is now "intelligent", since there is really no "outside" backtracking.

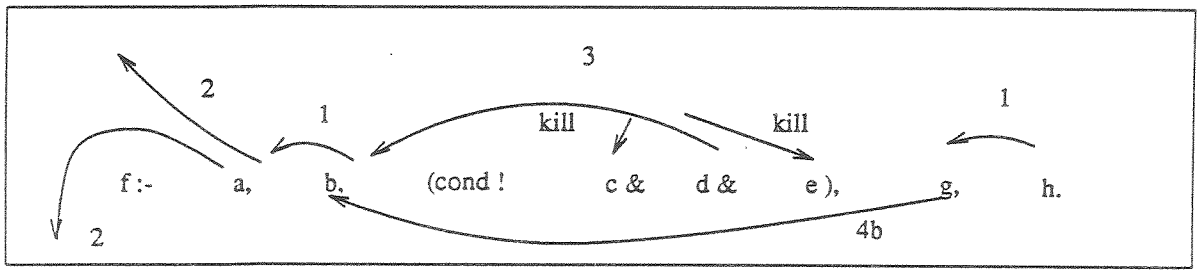

Figure 4-3: Backtracking cases for a CGE: Determinate Execution-(b)

A slightly different form of annotation is also sometimes useful:

`f(..):- a(..), b(..), (<conditions>! c(..) & d(..) & e(..)), g(..), h(..).`

Here the "!" inside the **CGE** makes *only the* **CGE** determinate. This annotation is used to take advantage of the efficiency of a determinate parallel call if it is known that **c**, **d**, and **e** will provide at most one solution path[29] (or if we are only interested in the first one they might provide). All cases of backtracking remain the same except case 4: since now the choices before the **CGE** are not "cut", case 4**b** is now (figure 4-3)

[29] An analysis of determinacy in the program can prove very advantageous in cases such as this: if the compiler can determine that **c**, **d**, and **e** are determinate (i.e. they will generate only one solution path) advantage can be taken of the efficiency of a determinate parallel call without the need for user annotations such as the one above. Such an analysis has already proved useful in limiting the generation of "choice points" in sequential systems [24].

"Outside" backtracking:

- **Case 4b-** If **g** fails, execution returns to the first choice point *before* the **CGE**. Again, no backtracking ever needs to be done inside the **CGE**. The only action required is to undo the bindings done by the goals in the **CGE**.

If the "cut" appears inside the **CGE**, as, for example, in

`f(..):- a(..), b(..),` **(<conditions> | c(..) & d(..) & ! & e(..))**`, g(..), h(..).`

all goals are executed in parallel as usual but, *upon exit from the* **CGE**, goals in the **CGE** which returned "with alternatives" and appear "to the left of" the cut in the **CGE** are marked as having "no alternatives"[30]. Of course the cut is also extended as usual up to the goal which called **f**. In this way goals which are "to the right of" the cut will still be backtracked using the "outside" backtracking algorithm, but, if they run out of alternatives, the backtracking point will correctly be before the calling of **f**.

Finally, if the "cut" appears outside the **CGE**, but there is another goal between the "cut" and the **CGE**, as in

`f(..):- a(..), b(..),` **(<conditions> | c(..) & d(..) & e(..))**`, g(..), !, h(..).`

then the **CGE** is executed normally (with no determinate optimizations, since the failure of **g** could cause "outside" backtracking) and the choices (goals with alternatives) in the **CGE** kept until the "cut" is encountered.

4.1.3.3. A General Algorithm

In the above, we presented a backtracking algorithm for clauses with embedded **CGE**'s through the use of examples. The general algorithm for non-determinate execution can be described as follows:

---

[30] In practice a *determinate call* is issued to these goals, which saves communication and computation overhead and makes scheduling and memory management more efficient. These subjects will be addressed in the next chapters.

- *Forward Execution: During forward execution leave a choice point marker (CPM) at each clause which still has alternative clauses which can be tried, and a parallel call marker (PCM) at each CGE which evaluates to true (i.e. each CGE which can actually be executed in parallel). Mark each PCM as "inside" when it is created, trigger the parallel resolution of the CGE goals, and change the PCM mode to "outside" when all those goals report success. Also, at this point, if the PCM contains only determinate calls, delete the PCM.*

- *Backward Execution: When failure occurs, find the most recently created marker (PCM or CPM). Then:*

    - *If the marker is a CPM, backtrack normally (i.e. as in sequential execution) to that point.*

    - *If the marker is a PCM and its value is "inside", cancel ("kill") all goals inside the CGE, fail (i.e. recursively perform the Backward execution).*

    - *If it is a PCM and its value is "outside", find the first goal, going right to left in the CGE, with pending alternatives which succeeds after a "redo", and then "restart" all goals in the CGE "to its right" in parallel. If no CGE goal is found to succeed in this manner, fail (i.e. recursively perform the Backward execution).*

This algorithm also turns out to be straightforward to implement at the abstract machine level. This will be clear when we present the implementation scheme in the following chapters. Other special cases will be covered then. In particular we will see how backtracking in the case where some of the goals which could have been executed in parallel are executed locally in a sequential way (e.g. due to a lack of resources) also fits within the same scheme.

4.1.3.4. Point Backtracking vs. Streak Backtracking

We call the algorithm for "outside" backtracking described in the previous sections "*point backtracking*": at any point during "outside backtracking" the algorithm looks for a *point of success* (i.e. a goal which responds with success after it is reinvoked) and only after such a point is found are the goals "to the right" of it restarted in parallel.

An alternative to "*point backtracking*" is "*streak backtracking*". In "*streak backtracking*", as the **CGE** is scanned right to left looking for the *point of success*, all goals which are encountered before a goal with alternatives is found for reinvocation are restarted in parallel with this last goal. The "outside" backtracking algorithm for *Streak Backtracking* is then (still referring to figure 4-1):

"Outside" backtracking: ("Streak method")

- **Case 4-** We have already finished executing all goals inside the **CGE** -we are **"outside"** the **CGE**- and we fail, having to backtrack into the expression. This is the case if **g** fails. Again processes not only report eventual goal resolution success, but also whether unexplored alternatives still remain for this goal.
    - If **g** fails and none of the **CGE** goals has unexplored alternatives, we will backtrack to **b** just as we would in the sequential execution model.
    - If **g** fails and one or more **CGE** goals still has unexplored alternatives, again our object will be to establish a methodology whereby all the combinations of those alternatives will have a chance to be explored, if needed, before we give up on the whole **CGE** and backtrack to alternatives prior to it. In *streak backtracking* the methodology used is to restart the parallel evaluation of all goals without alternatives up to the first one with alternatives (again scanning the **CGE** *in reverse order*, i.e. right to left), in parallel with the reinvocation of this last goal. This reinvoked process will then, in turn, report either **success** (with or without pending alternatives) or **failure**.

- If **failure** is reported, we continue as above: this goal and the following ones are restarted in parallel, up to the next goal with alternatives. Of course when a **failure** is reported by the leftmost goal with alternatives in the **CGE**, we give up on the whole expression and backtrack as in **Case 1** in previous sections (normal backtracking). Note that now the *streak of restarted processes* left during the execution of the algorithm has to be "**killed**" (the bindings created by the processes involved undone) before backtracking to the point beyond the limits of the **CGE**.

- If **success** is reported (i.e. a **point** of success is found in the **CGE**) no further action is needed: since the evaluation of goals to the right of the succeeding one was already started during backward execution, execution simply continues with the next goal to the right of the **CGE** as soon as this evaluation is completed.

Clearly, streak backtracking provides more parallelism, since the evaluation of the "backtracking point" (rightmost goal with alternatives) is done in parallel with the goals to its right. However, if all the goals with alternatives inside the **CGE** fail after being reinvoked, then all the work done by the *streak of processes* is not useful and has to be undone. Therefore, streak backtracking only appears to be of advantage if there are spare resources in the system. Except where otherwise noted, the following chapters will always refer to *Point Backtracking.*

4.1.4 Correctness of Conditional Graph Expressions

In a system based on goal independence, for a given clause or program, and procedural semantics

- A set of annotations, is defined to be *correct* if it only allows parallel execution of goals which are independent at run-time (i.e. if it guarantees that any parallel execution generated as a result of its evaluation does not result in variable binding conflicts) for any possible query.

- Also, a set of annotations is defined to be *complete* if it is capable of exploiting at run-time all possible AND-Parallelism which is *correct* (i.e. which does not generate variable binding conflicts) for any given query.

The definitions above can be relaxed in various ways. For example, correctness and/or completeness can be determined *for a given set of queries*, rather than for *all* possible queries. As an example, in the Delta-Prolog clause of the previous section, the set of annotations

```
f(X, Y, Z) :- ( a(foo1, X) /  b(foo2, Y) ), c(X, Y, Z) .
```

is *correct* for all queries of the form

```
?:- f(X, Y, Z).
```

where

a) `X` *and* `Y` *are ground terms (i.e. they contain no variables)* **or**
b) they are independent terms (i.e. they have no variables in common).

i.e. there are no variable binding conflicts possible in the parallel execution of `a(foo1, X)` and `b(foo2, Y)` for the defined set of queries. However, these annotations are not *complete* for the same set of queries: the following query

```
?:- f(constant1, constant2, constant3).
```

belongs to the set (the first and second arguments are ground), but the annotations would not generate all the possible parallelism, since *all* goals in the clause above could now run in parallel.

The conditions which guarantee *correctness* for a given **CGE** *for any possible query* are simple and follow directly form the definition of **CGE**'s and *goal independence*. Given a **CGE** of the form

(<CONDITIONS> | goal1(SV_1) & goal2(SV_2) & ... & goalN(SV_n))

where SV_i ($1 \leq i \leq n$) represents the set of *variables* in `goali` (or, if `goali` is itself a **CGE**, all the variables in all goals contained in it), we define two new sets of variables, $SV_\cap$ and SV_Δ where

$SV_\cap$ = { all variables which are in at least two of SV_1 ... SV_n }

and

SV_{Δ} = { all variables which belong to only one of SV_1 ... SV_n }

The *sufficient* condition for the **CGE** above to be *correct* is that its **<CONDITIONS>** field be composed of the conjunction of the following two conditions[31]:

- **ground(** $SV_{\cap}$ **)**
- **indep(** SV_{Δ} **)**

Intuitively, in order to prevent variable binding conflicts, all variables which are shared by at least two goals ($SV_{\cap}$) need to be ground at run-time, and all non-shared variables (SV_{Δ}) need to be independent. As an example, consider again the clause:

```
f(X, Y, Z) :-  a(foo1, X),  b(foo2, Y), c(X, Y, Z) .
```

The *correct* **CGE** annotation for parallel execution of **a** and **b** for any possible query is ($SV_{\cap} = \{\emptyset\}$, $SV_{\Delta} = \{X, Y\}$):

```
f(X, Y, Z) :-  (indep(X, Y) |  a(foo1, X) & b(foo2, Y) ), c(X, Y, Z) .
```

Similarly, the *correct* **CGE** annotation for parallel execution of all goals in the body of the clause for any possible query is ($SV_{\cap} = \{X, Y, Z\}$, $SV_{\Delta} = \{\emptyset\}$):

```
f(X, Y, Z) :-  (ground(X, Y, Z) |  a(foo1, X) & b(foo2, Y) & c(X, Y, Z) ).
```

Note that using the correctness conditions introduced in this section, once a set of goals is selected from a clause as candidates for parallel execution, the correct **CGE** can be determined automatically. Thus, these conditions, coupled for example with some heuristics for the selection of goals (perhaps guided by some user

---

[31] It is difficult in general to express *correct* Graphs using DeGroot's syntax because *conjunctions* of conditions are not explicitly allowed.

annotations), can be used as a starting point in the design of an automatic CGE generator.

4.2 Programmer's View of the RAP System

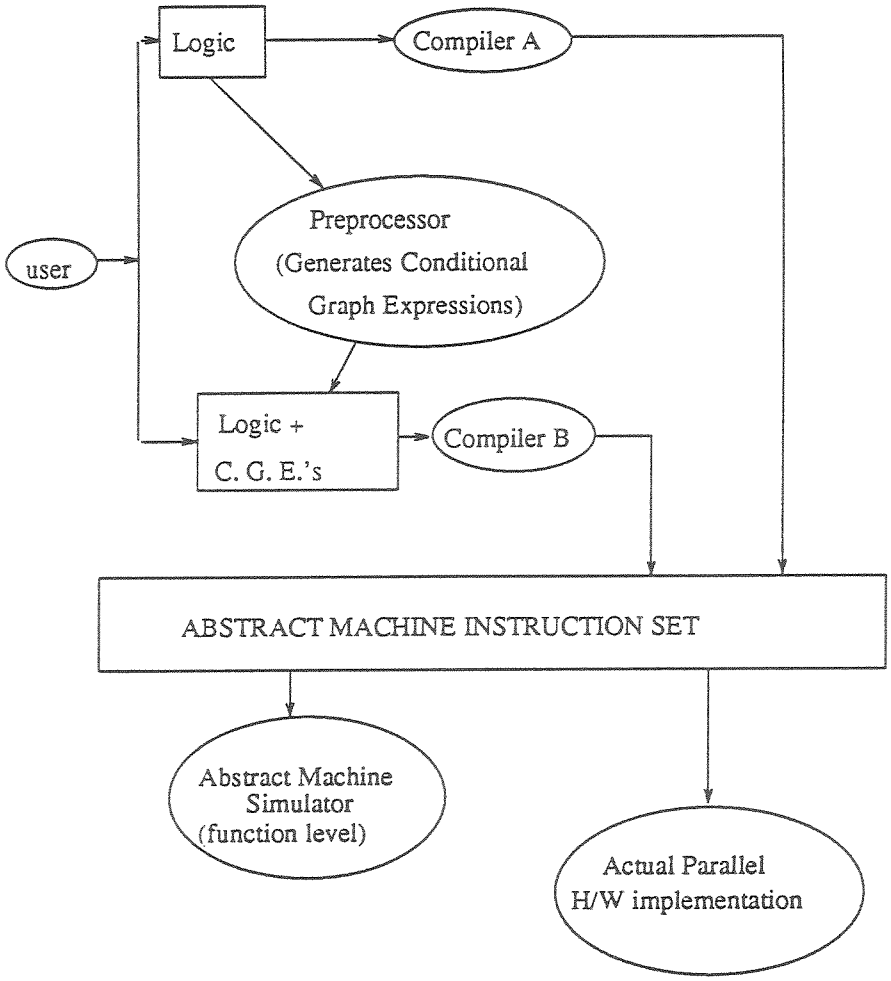

Figure 4-4: Programmer's View of the RAP System.

As stated before, the main objective of the following chapters will be to address the design of an *efficient* execution model for the parallel implementation of Logic Programs, capable of supporting AND-Parallelism in the presence of "don't-know" non-determinism. It was also mentioned how the emphasis would be, rather than on a particular language, on developing techniques which can be applied to a

variety of languages that include this type of non-determinism. However, it is also interesting to at least consider the implications which a model such as the one described in this chapter has at the user's level.

In this sense, an interesting objective to be fulfilled by goal independence models which was already pointed out in previous chapters, is to provide an implementation scheme which preserves execution efficiency (so that run-time overhead does not overshadow the expected performance improvement) while keeping as much as possible the issues related to parallelism transparent to the programmer. Figure 4-4 represents a system based on Restricted AND-Parallelism from the programmer's (user's) point of view. Two alternative implementations are suggested: in alternative A, an intelligent compiler ("compiler A") would perform the necessary analysis of the source program (written perhaps in Prolog) translating it into instructions for an abstract machine capable of supporting Restricted AND-Parallelism, that is, a machine supporting independence checks and control of parallel processes as well as sequential execution. The generation of **CGE**s would then be one of the early steps of the compiler, and the **CGE** format just an internal compiler representation.

Clearly, the arrangement described above does succeed in hiding control issues from the programmer at the source code level. Nevertheless, there has always been controversy in the logic programming community (and many others) with respect to whether or not the user should actually also be provided with mechanisms for expressing information about parallelism and data dependencies in the form of annotations at the source program level. As stated in previous chapters, it is felt that annotations hurt the otherwise clean declarative semantics of logic programs and that the user should be shielded from the issues that annotations address (i.e. parallelism,

control, etc.). On the other hand sometimes the user has vital information concerning these issues readily available that would be very painful for the compiler to extract from only the set of clauses in the source program. This is the case, for example, when the user knows exactly the type of queries which will be presented to the system and is more interested in performance than in flexibility of the program ("input mode" or "invocation mode" annotations).

In many cases this controversy can be easily settled by providing support for both types of code (annotated and non-annotated) at the same time. This is specially easy in the case of **CGE**'s since the declarative semantics of the annotated code is equivalent to that of the original clause (if the **CGE** is *correct*). Such an approach is presented as alternative B, referring again to figure 4-4. This alternative is based on observing that the process of generating **CGE**'s for a program, which can always be considered an independent phase of the compiler, can in fact be completely detached from the compiler itself. This step can then be viewed as a preprocessor that turns a standard program into an annotated one. A less sophisticated compiler ("compiler B") takes care then of the translation from the annotated program to the abstract machine level instruction set. This is the exact situation shown in figure 4-4. The programmer can then write non-annotated programs, annotated ones or a combination of both. This approach has the additional advantage that the results of the preprocessing are incorporated to the source program in terms that are understandable to the programmer. In this way, the programmer can view or modify the results of the preprocessing if it is necessary for optimization purposes. Programming with annotations can be envisioned as "expert level", while non-annotated programs would represent a more "naive user level".

4.3 Chapter Summary

This chapter has presented and defined forward and backward execution algorithms for Horn clauses with embedded Conditional Graph Expressions (**CGE**'s). First, these expressions were introduced and their correctness and completeness defined. Sufficient conditions for proving the correctness of a given **CGE** were also provided. It was shown how conventional backward execution is not applicable to **CGE**'s and several backward execution algorithms were provided for dealing with determinate and "don't know" non-determinate execution. Therefore, the model presented can support "don't know" non-determinism in the presence of AND-Parallelism. Finally, a view of such a model from the programmer's perspective was discussed.

Chapter 5

A High-Level Execution Model for AND-Parallelism: Memory Management and Goal Scheduling

In the previous chapter the procedural semantics for an AND-parallel execution model for Logic Programs were presented. However, there are several other issues which have to be addressed in order to complete such a model, and which very dramatically affect its *efficiency* in a practical implementation. Two such issues are *goal scheduling* and *memory management*. In this chapter, the relationship and interactions between these two issues will be studied. Basic scheduling and memory management strategies will be presented, and the implications of their implementation on the overall performance of a system will be analyzed, where the desirable characteristics to strive for are: minimization of idle processor time, memory usage optimization, garbage collection minimization, and load balancing, among others. Although, reference will also be made to other cases such as deterministic models (as in functional languages) and "don't care" non-deterministic models (committed choice systems) the study will concentrate on the particular issues involved in the implementation of the "don't know" non-deterministic model presented in the previous chapter. It will be shown how the techniques used in sequential systems for avoiding garbage collection through the recovery of space during backtracking can be extended to a parallel, distributed stack system.

5.1 A Simplified Model of Logic Programming Implementation

Figure 5-1 shows a *very simplified* memory management model for a typical stack-based implementation of Logic Programs. Although a realistic model, such as the Warren Abstract Machine (WAM) [88], includes several stacks (for "environments", "choice points", local and global data structures, "trailed" variables, etc.), in our discussion we will reduce this model to a single stack which simply contains activation records (AR's), one for each invocation of a goal. We will introduce a more complete model in the next chapter when we deal with abstract machine level issues.

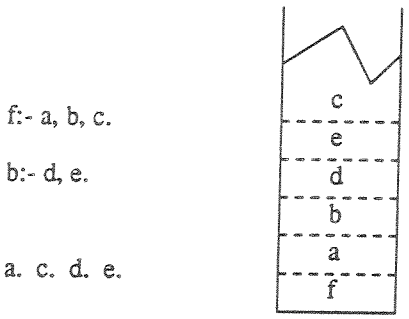

Figure 5-1: A Single Stack Model

We will also suppose in this first model no optimizations (such as "last call" optimization [87] and others, to be introduced in the next chapters) so that these activation records will in general stay on the stack even after successful return from a procedure. As will be shown in the following chapters, this is not an unrealistic approximation since, even if such optimizations are implemented, the *single* stack still has to contain all data structures as well as the conventional "local stack" entries. Thus, the remaining activation records in the single stack model represent the pending data entries which would still remain in the "global stack" and "trail" of a

conventional system after last call deallocation. With these premises, depth-first execution of the set of rules in figure 5-1 would leave a "trace" of activation records in this single stack as shown in the same figure.

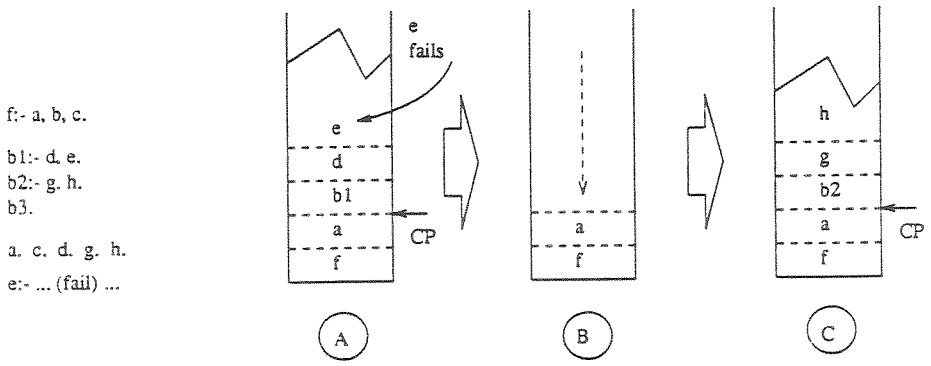

Figure 5-2: Backtracking in the Single Stack Model

Deterministic execution (i.e. execution of a program section where there is only one possible matching clause for each goal invocation step, as in figure 5-1), is fairly straightforward in this model: the stack simply grows with each invocation (call) until a final failure or success occurs, or until memory space is exhausted. In this last case, garbage collection is necessary in order to continue.

A more interesting behavior occurs in non-deterministic cases, for systems which implement backtracking. Consider the example shown in figure 5-2. In this case there are three alternatives for **b**: $\mathbf{b_1}$, $\mathbf{b_2}$, and $\mathbf{b_3}$. Supposing depth-first execution, the call to **b** in the body of **f** matches first with $\mathbf{b_1}$. **d** and **e** (figure 5-2-A) are executed subsequently. Suppose now **e** fails: the next alternative, $\mathbf{b_2}$, will have to be considered, and all results (bindings, new data structures, environments etc.) related to the failed invocation of $\mathbf{b_1}$ should be discarded. This is done in all practical implementations by trimming the now invalid top portions of all stacks (bindings are normally undone while "trimming" the top of a special stack - the Trail). This

process is represented in our simplified model by discarding the top of the single stack, as shown in figure 5-2-B. Execution can now proceed with $\mathbf{b_2}$ just as with $\mathbf{b_1}$ before, resulting in the stack pattern represented in figure 5-2-C. Clearly, the main advantage of this mechanism is the complete retrieval of all used space during backtracking. In fact, backtracking is extensively used by programmers in practical systems for the purpose of avoiding garbage collection.

The single stack model presented above really only reflects the relative *precedence* of AR's in the stack, as a function of the order of execution of the goals. However, much of the efficiency of current implementations depends on this relative ordering to be able to perform the space retrieval operations on backtracking described above. As we will show in the following sections, when the single stack shown in figure 5-1 is unfolded into a set of stacks during parallel execution, this relative ordering will not only depend on the procedural semantics of the language, but on many other implementation-dependent parameters. A simple extension of the simplified model presented above will help us keep track of the ordering of data inside the multiple stacks as a function of those parameters, with limited regard for the details involved in a more realistic implementation.

5.2 Towards Parallelism

The scheme presented in the previous section is only suitable for modelling the behavior of *sequential* execution schemes for logic programs. However, it can be easily extended to model most stack-based parallel execution models, i.e. those which are based on extending the techniques used in sequential systems by implementing a distributed stack system. Some distributed stack logic programming systems are described in [55], [31], [89], [7], and [12]. One of the main reasons supporting a stack-based approach is the fact that, although logic programs can present considerable

opportunities for parallelism, there are always (determinate) code sections requiring sequential execution [77]. A system which can support parallelism while still incorporating the performance optimizations and storage efficiency of current sequential systems is thus highly desirable. Stack-based systems still seem to be at present the fastest and most efficient models for logic program implementation.

5.2.1 A Simple, Distributed Stack Model

Let us suppose a "generic" parallel system architecture, as shown in figure 5-3. In general, we will assume the existence of a number of processors and a number of memories and that all processors have access to all memories through some kind of interconnection network. This access could actually be direct (as in the case of a global shared memory), switched, or through messages. For simplicity, direct access with some kind of global addressing scheme (so that system-wide references are possible) will be supposed from now on. In addition to processor to memory access, the interconnection network also provides communication via message passing from each processor to each of the other processors. Again this can also be emulated trough common memory, but for simplicity a message passing capability will be assumed.

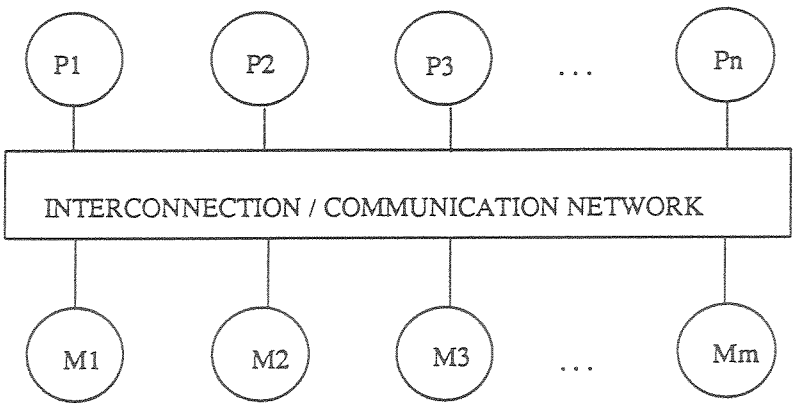

Figure 5-3: System Architecture

In order to model the distributed-stack, parallel implementation of a logic program on such an architecture, we will suppose that the program itself comprises sequential sections which eventually arrive at a point where several execution paths can be taken simultaneously (a "fork"), and that the control of this "forking" behavior is determined by annotations, in particular by Conditional Graph Expressions and their related algorithms as presented in the previous chapter. As an example, consider the following clause:

```
f(X,Y,Z) :- a(X,Y), (ground(X,Y) | b(X) & c(Y,Z)), d(X,Y,Z).
```

where, as described in the previous chapter, the presence of the Conditional Graph Expression (CGE) `(ground(X,Y) | b(X) & c(Z))` determines that during the execution of `f`, `a` has to be executed first, and then `b` and `c` can be executed in parallel if `X` and `Y` are determined to be ground at run-time. `d` will have to wait for all of them to finish in order to start its execution.

The execution of a parallel AND-"branch" (such as that described above) in our simplified distributed stack model is represented in figure 5-4-A.

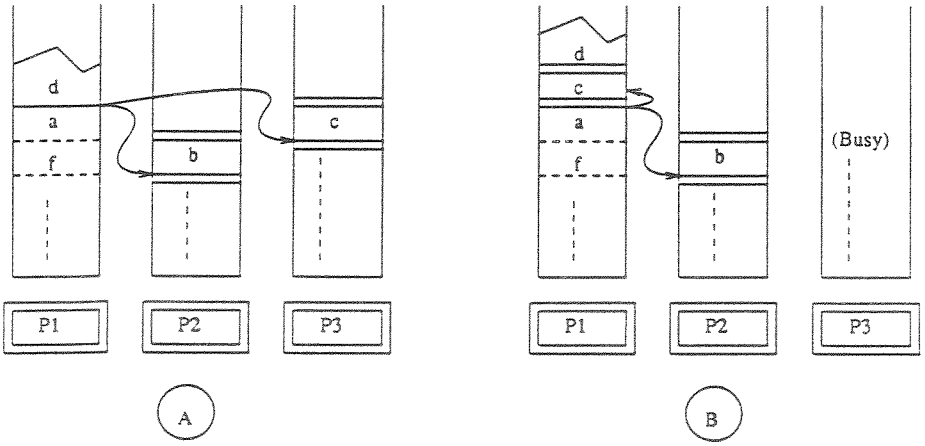

Figure 5-4: Distributed Stack Execution

In this case, execution of **f** starts in *P1*'s stacks (the part of common memory assigned to *P1*), but as **a** succeeds, goals **b** and **c** are executed remotely and in parallel in *P2* and *P3*. All new data and control structures created by the execution of goals **b** and **c** are located in their respective stacks. Note that since the goals are determined to be independent by the checks, no variable binding conflicts will occur: *P2* and *P3* may *read* any values from *P1*'s stacks (e.g. the values of **X** and *Y*) but at most one of them will write into these stacks for each particular variable (e.g. in this example only **c** will write the value of Z). In most implementations, such as the one which will be described in the next chapters, the value of Z would be *constructed* in *P3*'s stacks, and only a *pointer* to this value would be written into *P1*'s stacks[32]. Furthermore, in a shared memory system, communication traffic is minimized by not transmitting these newly created structures back to the "parent" (*P1*), since they can be readily accessed in their current locations. When **b** and **c** finally succeed, execution of **d** can continue in *P1*, as shown in figure 5-4-A. What was a single stack in the sequential model is now unfolded into a "distributed" stack, scattered across the memory areas corresponding to different processors[33]. An alternative execution of the clauses in the example above is shown in figure 5-4-B. In this case, after the success of **a**, execution of **b** starts as before in *P2* but now, since *P1* is idle (execution of **d** has to wait for **b** and **c** to succeed), it starts executing **c** itself, thus leaving *P3* free to perform some other task. When both **b** and **c** have succeeded, execution of **d** continues in *P1*.

Again, this greatly simplified distributed stack model only reflects the

---

[32]However, if Z is instantiated to a constant, the value of the constant itself would be written into *P1*'s stacks.

[33]Borgwardt has also proposed a distributed stack model [7] although *stream AND-Parallelism* guided by input-mode annotations (rather than *restricted AND-Parallelism*) is supported.

relative *precedence* of AR's in the stack. Many other details have been left out purposely in order to concentrate on the main issue of this chapter: the relationship between goal scheduling and memory management in a distributed AND-Parallel system. The lower-level details of the model will be given in the following chapters, where abstract machine level implementation issues will be presented. Also, the model does not show the additional complications introduced by *pure* OR-Parallelism: in an OR-Parallel system the distributed stack is really a *tree of stacks* [12] [7] [55] . However, the model will show in a simple way the factors which affect the relative ordering of AR's: this ordering will not only be a function of the ordering of goals in the source program, but will also depend on the goal scheduling strategy. In the next sections we will first review some *distributed goal scheduling strategies* useful for the assignment of goals to processors in a parallel logic programming system. We will then propose *process(or) state diagrams* and *stack management schemes*, using the model above to determine goal precedence and distribution across processors. This information will be useful in the assessment of the viability and efficiency of the schemes proposed. Although most of the considerations in the following sections also apply to processes in a system supporting multiprocessing, for simplicity we will generally refer to "processors" from now on.

5.2.2 A Simple Goal Scheduling Strategy

When following the solution of **f** in the previous example, given that **f** is being executed in a particular processor (*P1* in figure 5-4) and it arrives at a point where several goals are available for parallel execution (**b** and **c** in the same figure), the question arises as to how which policy is to be used in order to distribute these goals to the free processors in the system. This distribution is herein referred to as "goal scheduling" and the policy used the "goal scheduling strategy".

A possible goal scheduling scheme is to have the processor which encounters

goals which can be executed in parallel look for idle processors, assign one of these goals to each of the idle processors, and continue executing one of the remaining ones itself. The problem with this scheme is that in it all the "scheduling duties" (looking for idle processors, sending goals, etc.) are performed by a processor which already has work to do, and thus the time involved in performing them adds up as overhead. It is in general a better idea to put this burden in the hands of otherwise idle processors. The following is a very simple and completely distributed scheduling strategy, in which idle processors "steal" goals from busy processors:

- Each processor has a private "goal stack", which is initially empty.
- All processors are initially idle. The user query is placed in the goal stack belonging to one of the processors. Execution starts in this processor and with this goal.
- On arriving at a point where several goals are available for parallel execution, those goals are also pushed on to this goal stack.
- Goals can be picked up from such goal stacks for execution both by the owner of the stack (the processor which loads goals on to it) and/or by any remote processor. A remote processor picks up goals in the following way:
    - An idle processor looks into other processor's goal stacks, until it finds one that is not empty. Then, it "steals" a goal from that stack and starts working on it. When the execution of that goal is finished, the result (success or failure) is reported to the processor the goal was taken from.

The first processor thus starts working on the first goal, and as it pushes goals into its private goal stack they are picked up by other processors. They will in turn generate other goals to be picked up by other processors and work spreads itself in this way naturally over the network as it becomes available. Note that this algorithm is valid even if there is only *one* processor present (or not faulty) in the system: since any processor can also pick up goals from its own goal stack, all goals scheduled for execution in parallel may also be executed in this only processor.

An obvious optimization can always be applied to this scheduling scheme: in the algorithm above a processor always pushes *all* the available goals on to its goal stack, probably only to immediately pop one of them locally to continue working on it. Instead, it can routinely push all but one, for example the last one, and execute this last goal right away without needing to go through a push-pop sequence in the goal stack. Thus the time involved in the push-pop sequence is always saved for at least one goal.

As pointed out above, in this system scheduling duties are performed by otherwise idle processors, rather than by busy ones where these duties would add up as overhead. Also note how scheduling is completely distributed and thus the system is scalable (if the interconnection/communication network scales well too).

Other related distributed load balancing schemes have been proposed. For example, Keller et al. [38] propose a load balancing scheduling strategy in the *Rediflow* architecture. Their scheme is intended for distributing load ("pressure", in their model) as uniformly as possible using a mesh topology as an example. Processors have queues which represent a reservoir of "runnable" processes (*Chares*). A processor evaluates its internal "pressure" (measured as the number of entries in the queue, but affected by some other factors) and compares it with that of its neighbors. When the local pressure is comparatively high some "Chares" are issued forth into the interconnection network, where they are distributed to less loaded processors. Similar schemes have also been proposed by Burton and Sleep [8].

The main problem with the schemes above is that when the number of processors is large, "polling" from one goal stack to another in order to find available work can be very inefficient. Some global scheduling mechanism, capable of pointing

idle processors to available work would be desirable, but care must be taken to prevent this mechanism from becoming a serial bottleneck in the system. Such a mechanism is presented below.

5.2.3 A More Efficient Goal Scheduling Strategy

Consider the addition of a new element to our basic system architecture (figure 5-3) in the form of a "scheduling network" (figure 5-5).

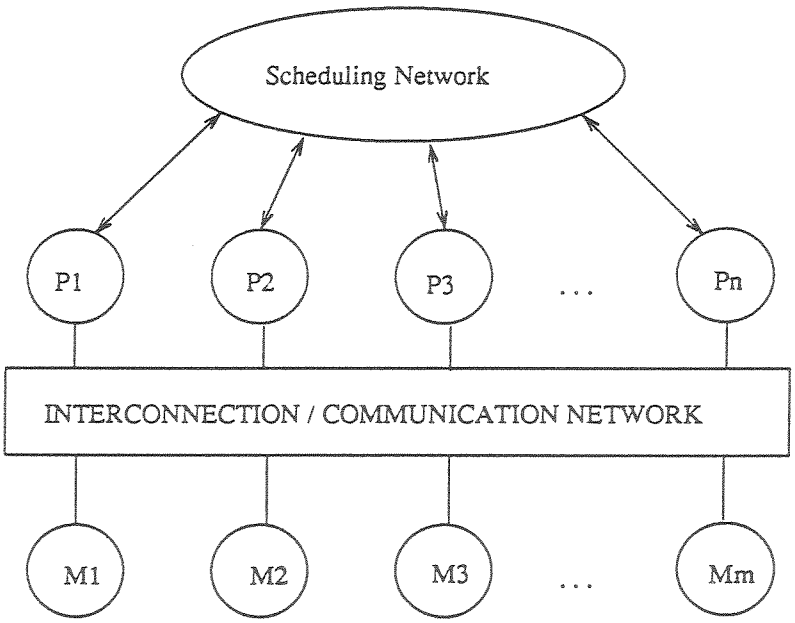

Figure 5-5: System Architecture

This element is connected to all processors, and it acts as a global scheduling mechanism. Its operation can be summarized as follows:

- Each processor continuously feeds a **value** into the network which can represent either its load (for example the number of processes running if multiprocessing is supported) or the amount of work available in it for other processors (for example the number of goals in its goal stack).

- At any point in time any processor can ask the network for the Id. of the processor feeding in the highest **value**, and this Id. will be provided by the network with very little delay.

Once again, in a shared memory system this function can be emulated by a globally accessible (and constantly updated) value in common memory, but we will suppose the existence of the network for clarity. Such a scheduling network can be implemented by anything from a wire-or bus (which would provide one maximum value) to a tree structure (such as a sorting network [5], which would provide the N maximum values), with different delays and cost depending on its complexity. In particular, the delay and cost can be kept sublinear (logarithmic) whith respect to the number of nodes (processors) involved. The following algorithm is an extension of that in the previous section but making use of a scheduling network such as that described above:

- Each processor has a private "goal stack", which is initially empty.
- All processors are initially idle. The user query is placed in the goal stack belonging to one of the processors. Execution starts in this processor and with this goal.
- On arriving at a point where several goals are available for parallel execution, those goals are also pushed on to this goal stack.
- Goals can be picked up from such goal stacks for execution both by the owner of the stack (the processor which loads goals on to it) and/or by any remote processor. Goals are picked up by remote processors in the following way:
    - The number of goals in the private stack is continuously fed by each processor to the scheduling network[34].
    - An idle processor receives from the scheduling network the Pid. of the processor with the highest number of goals in its private goal stack. It then "steals" a goal from that stack and starts working on it. When the execution of that goal is finished, the result (success or

[34] As pointed out before, if multiprocessing is implemented, then the number of processes being run should also be used. These two numbers (available goals and processes currently being run on the processor) can be combined to provide a total **load** number which is fed to the network. The scheduling scheme would then ensure that idle processors always picked up work from the most heavily loaded processors.

failure) is reported to the processor the goal was taken from (parent processor).

The time spent by processors polling other processors is now eliminated, and the delay between a processor being available and it finding a goal when there is available work in the system is minimized and dependent only on the scheduling network delay. Thus the the only scheduling bottleneck is the (sorting) network itself, but its action is limited to a very simple operation whose delay can be kept sublinear. This system also achieves "load balancing" which is useful if interprocessor distance (measured as communication delay) is constant. If this is not the case, then other strategies that make use of locality can be implemented: for example, the scheduling network can compute a value that is not simply the maximum of the numbers being fed in (as above), but a function of the number fed in by each processor and the distance from this processor to the one issuing the request for a goal. The Pid. obtained then is not only that of a processor which has many goals available, but also that of that of one which is "close" to the requesting one. The particular function used would obviously depend on the topology of the network.

Since more than one processor could attempt to pick up a given goal simultaneously, goal stacks obviously have to be locked during this operation. In order to prevent many processors from fighting for access to the goal stack with the maximum number of goals, a simple optimization can be introduced: while the goal stack of a processor is locked, the value fed to the scheduling network will be zero. Thus this processor will not receive requests from others until it is free again.

5.2.4 A Simple Processor State Diagram

Figure 5-6 shows the states which a process or processor, running in a system implementing the scheduling strategies described above, has to go through, from initialization to success or failure of a given query. Consistent with normal state transition diagram conventions, states are represented by circles and transitions by arrows. Transitions are caused by the reception of messages. These messages and the outputs generated by the transitions are given associated to the arrows. Messages enclosed in quotes ('...') are **inputs** from the *scheduling network*, those preceded by / are **outputs** to the *communication network* and the rest are **inputs** from the *communication network*.

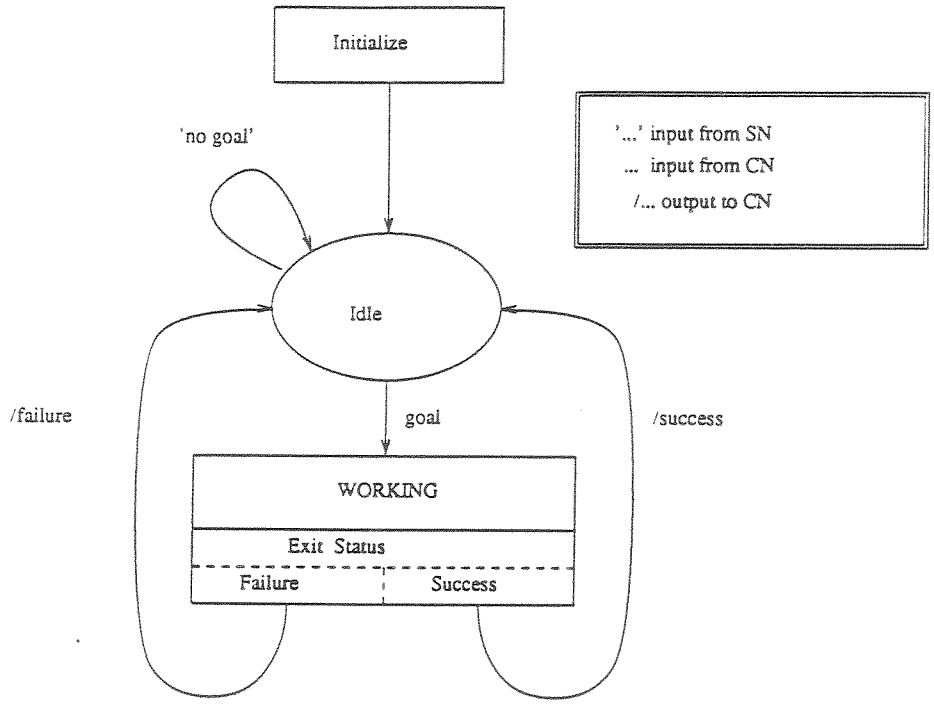

Figure 5-6: Simple Process State Diagram

The diagram describes the actions performed by a processor until a goal is received and work starts on it: after initialization, the processor goes into an idle

state where it continually asks the scheduling network for the Id. of a processor which has goals available in its goal stack (or it continually polls other processors in the simpler scheme). When a goal is received, the system goes into a complex state ("Working" in figure 5-6). In a non-shared memory system, two local processes would be started. The "foreground" process would be the one actually working on the goal, i.e. the process that is executing the user program. Its states would be determined by the different instructions being executed[35].

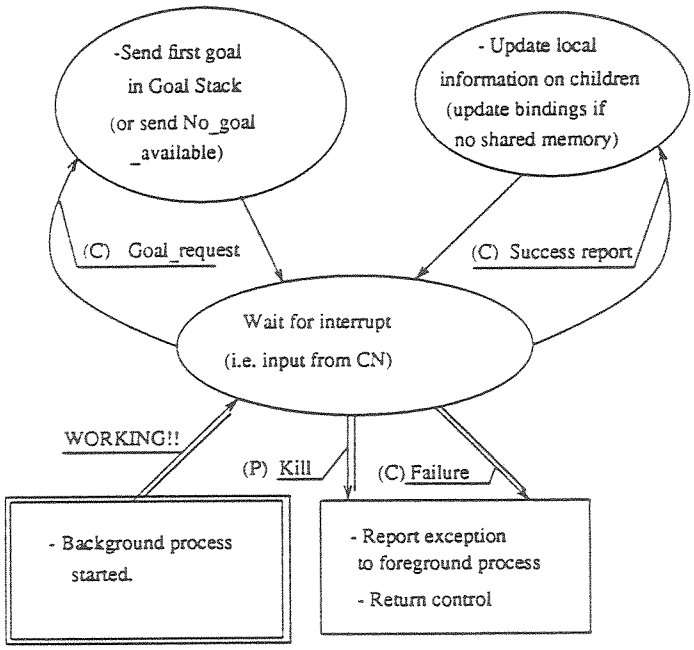

Figure 5-7: Background Process State Diagram

The "background" process would be in charge of sending goals from the local goal stack to remote processors upon receipt of a request, of updating local information regarding success of "children" (e.g. whether they still have alternatives

[35]These actions and instructions will be described in the next chapters.

or not), and of reporting exceptions to the foreground process (e.g. the reception of a "kill" message from the parent). A state diagram for the background process is shown in figure 5-7. Normally the background process would be terminated by the foreground process upon finishing work on the current goal (with either success or failure). In this scheme the background process is independent from the foreground process except for the fact that it can interrupt the latter to report two kinds of exceptions, as described below. After either of these exceptions the background process terminates itself.

The first exception occurs when a kill message is received from the parent process. The foreground process stops working on the current goal and it returns to the corresponding idle state (figure 5-6). The second kind of exception appears when failure is reported by some child of the current process. In this case execution continues normally but the foreground process handles the failure according to the procedural semantics presented in the previous chapter.

In practice, the most efficient way of implementing this dual process (background/foreground) behavior is through an interrupt mechanism. Note that the normal state of the background process is waiting for an interrupt, in the form of a message from the network (Goal_request, Success, Failure or Kill). This makes it a good candidate for an interrupt based implementation: if one of these messages arrives to the processor mailbox while executing a program instruction, an interrupt flag is set. At the end of the instruction the flag is sensed and the (microcode) routine corresponding to the particular type of interrupt (message id -one of the four above) is executed. "Goal_request" and "success_report" will return control after being serviced to the next instruction in the program. "Failure" will start the failure management (microcode) routine, and restart execution of the program at the

appropriate place (this will become clear when failure behavior is explained for the abstract machine in the next chapters). The "kill" management routine executes the kill semantics (i.e. kill all dependents) and branches to the idle loop.

Of course, in a *shared memory* environment there is really no need for a background process at all: when an idle processor asks the scheduling network for the Pid. of a processor having available goals, the value received can be the actual address of the top of the goal stack for that Pid. The goal can be picked up from there directly by the requesting processor and it can also update the memory areas of the parent processor to report success, provided proper memory arbitration is implemented. The messages arriving at the mailbox can be detected between instructions (i.e. as interrupts) and handled as described above. Again, work on the received goal in this only ("foreground/background") process will eventually finish in success or failure (see figure 5-6) and the processor will return to the idle loop, ready to start work on another available goal.

5.3 Memory Management and Scheduling

The simple, distributed stack memory management scheme introduced in section 5.2 can be combined with the above described scheduling algorithms and the processor state diagram in figure 5-6. The contents of the stacks during a sample execution of the clauses listed below in such a system are shown in figure 5-8; both the simple stack of section 5.2 and the goal stack introduced above are being represented.

```
f:- a, (b & c), d.
d:- (g & h).
```

Note how in figure 5-8, goals which are available for execution in parallel (such as **b** and **c** in figure 5-8-B) are pushed on to the goal stack of the processor

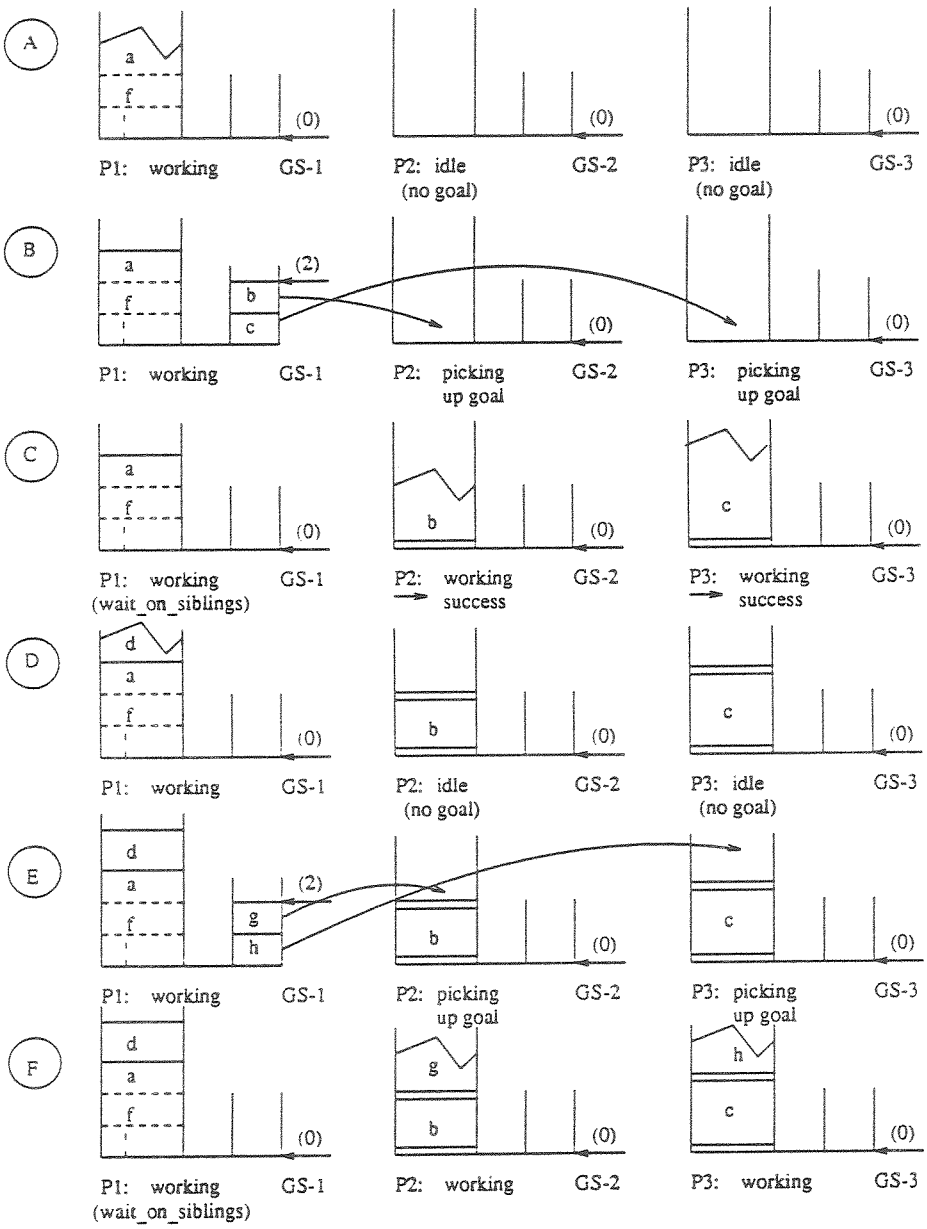

Figure 5-8: Goal Stack Based Goal Scheduling

which encounters them. From there they are picked up by free processors (figure 5-8-C) which work on them until they completely succeed. If these processors in turn generated new goals for execution in parallel, they would be pushed on to *their* goal

stacks and picked up from there by other free processors or, eventually, by themselves. In principle, after execution of a given goal is completed, a new one can be picked up and execution can proceed stacking all new data structures above the old ones[36] (figure 5-8-D,E,F). An "Input Goal Marker" (represented by double horizontal line in figure 5-8) is used for separating different *stack sections*, each one of them containing all structures related to the execution of a given goal, which was taken from another processor[37].

The scheme used in figure 5-8 can be used directly in "committed choice" systems. These include all stack-based implementations of functional languages and of logic languages which make use of "don't care" non-determinism, such as Concurrent Prolog [72], Parlog [30], and GHC [81]. Execution in these models could proceed as in figure 5-8, allocating all structures corresponding to the execution of new goals on top of those corresponding to previous ones. In the event of memory exhaustion, a (distributed) garbage collection algorithm will have to be used in order to retrieve unused space.

We have argued in previous chapters how "don't know" non-determinism as implemented in full OR-Parallel systems (such as [55]) and in backtracking systems (such as Prolog [58]) is of special interest in Logic Programming. We have also argued that even though OR-Parallelism can be interesting in the presence of spare resources, backtracking also has to be supported in such systems in order to prevent a combinatorial explosion of the search space and the number of processes generated. The procedural semantics of a system capable of supporting AND-Parallelism and

---

[36] See section 5.3.1 for a more detailed discussion on this subject regarding backtracking systems.

[37] Creation of these markers is only strictly necessary if backtracking is to be supported.

"don't know" non-determinism through distributed backtracking was presented in the previous chapter. In the next sections we will analyze memory management issues that particularly apply to don't know non-deterministic systems which make use of distributed backtracking, and the interaction between memory management and the goal scheduling strategy being used.

5.3.1 Memory Management Problems Associated with Distributed Backtracking

The basic condition necessary for efficient memory management in *sequential backtracking* systems is that at every point in the computation, *newer* structures always be stacked on top of *older* structures. If this rule is observed, then all used space can always be retrieved on backtracking and this retrieval is always done from the top of the "stack" (as in figure 5-2).

Note that in practice, "newer" and "older" have to be defined in terms of the particular control strategy being used, i.e., for Prolog "older" means "closer to the root" and "to the left of" in the depth first, left to right execution tree. In general, for backtracking systems, we will define a goal invocation **a** as being "older" than another goal invocation **b**, represented as $\mathbf{a} < \mathbf{b}$, if, for a given control strategy, *all alternative solutions of* **b** *are to be tried before a new solution of* **a** *is attempted*[38].

In parallel systems, equivalent conditions to those above are to be applied to the distributed stacks involved in execution. In particular, if the following two

---

[38]Note that although this relationship is always defined for any two goals in a sequential deterministic system such as Prolog (given any two goal invocations **a** and **b**, $\mathbf{a} < \mathbf{b}$ if **a** is executed before **b**) this is not always the case in some parallel systems and there may be cases where neither $\mathbf{a} < \mathbf{b}$ nor $\mathbf{b} < \mathbf{a}$ is true. However, this relationship does hold in all cases for the backtracking strategy defined in the previous chapter.

conditions are met, the efficiency present in sequential systems can be preserved in distributed backtracking systems:

1. The same precedence as in the sequential model is maintained within each stack section during the execution of each particular goal.

2. If stack sections corresponding to different goals are allocated on the same physical stack (as in figure 5-8-F, for goals **b** and **g**) then only structures which correspond to an *newer* (descendant) goal of the one currently on top of the stack are allocated above it.

These two conditions, added to backtracking algorithms such as those proposed in the previous chapter, ensure that, in the event of failure, backtracking in parallel systems will also be able to retrieve all stack space used in the computation of the failed alternative from the *top* of all stacks, and that execution of the new alternative will start from the new tops, so that there is free space above for execution to continue.

It should clear that, once a goal has started execution at any given processor, the *first* condition above, i.e. that newer structures always be stacked on top of older structures during the execution of that goal within a stack section, can be met simply by using the same techniques currently applied in conventional sequential models. This will be the approach taken in the Abstract Machine which will be described in the next chapters.

Let us therefore concentrate on the second condition above and start by analyzing the problems associated with *not* meeting it. Returning to figure 5-6, consider the situation in which a processor, having succeeded in the execution of a given goal, goes back to the idle loop to look for another goal to work on. This is the case, for example, of processor 2 in figure 5-8-D,E,F: after the success of **b**, a new goal

g is picked up. In order to illustrate a situation in which problems can arise, suppose that instead of **g**, a different goal **k** (from the execution of another parallel call somewhere else in the system) had been "picked up" and that **k** > **b** does not hold (i.e. **k** is not a descendant of **b**). In this case, when dealing with a failure at some point during the subsequent execution of the program *it is possible that* **b** *may have to be backtracked before* **k**. If, as a result of such backtracking, **b** needs to be "killed" (i.e. deallocated) this represents only a minor problem: deallocation of **b** is possible, leaving an empty slot of "garbage" in the stack, and execution can continue (figure 5-9-A: the **"garbage slot"** problem). Although this space may be retrieved later through backtracking (i.e. if **k** is deallocated before anything else is stacked above it) this cannot be guaranteed in general. Therefore, complete retrieval of used space during backtracking is not preserved.

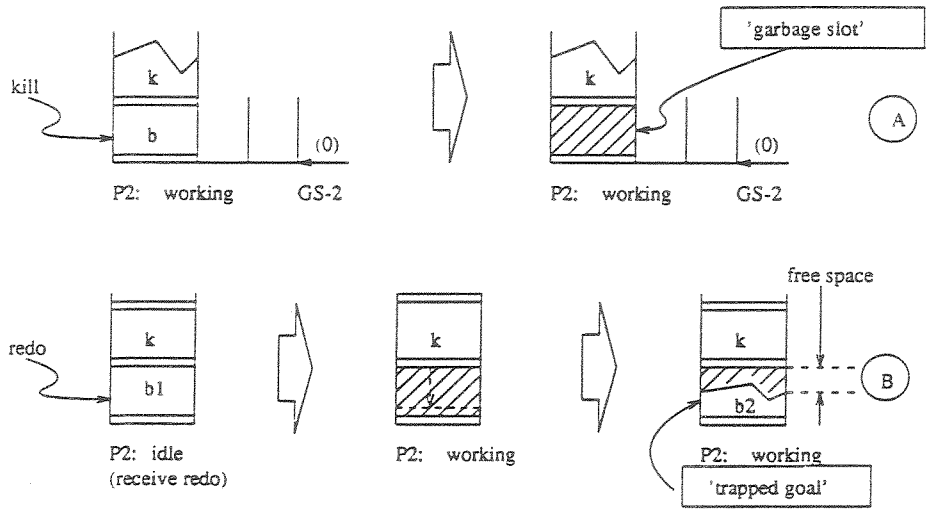

Figure 5-9: The "garbage slot" and "trapped goal" Problems

A more serious problem appears also during backtracking if, for the same situation depicted above, an alternative solution is needed from **b** (i.e. a *redo* message

is sent to *P2* referring to **b**) and again **k** has not yet been deallocated. In this case part of the stack space occupied by **b** will be deallocated (down to the next "choice point" - figure 5-9-B) and a new alternative will be evaluated, its structures being stacked above this point. But note how, since there is no a priori limit to the number of structures that will be needed in the evaluation of this new alternative, the space available in the stack below **k** could be insufficient for the complete evaluation of this new alternative (figure 5-9-B, the **"trapped goal"** problem). Using space *above* **k** would additionally complicate complying with condition 1 above. If, on the other hand, **k** were a *descendant* of **b** (as is the case of **g** in figure 5-8-F), *all alternatives of* **k** *would have been tried, and* **k** *itself deallocated before* **b** *is backtracked.* This would be the case if Prolog style semantics are applied to the example in figure 5-8.

Although in some models conditions 1 and 2 above can be met by forcing a precise ordering of events, we will be interested in this chapter in exploring more general solutions, those that can be applied to different models with some independence of the actual ordering of events in the distributed system. Therefore, in the next sections we will present a number of modifications to the basic scheduling and memory management model presented so far, in order to extend its application to general "don't know" non-determinate systems using backtracking while avoiding the problems illustrated in this section.

5.3.2 The Idle Processor Solution

One way to prevent stacking a non-descendant goal over an existing one is to avoid stacking goals at all. Of course, this is a rather trivial solution, but it will used as a base case upon which the other solutions proposed in the next sections will be built. Figure 5-10 shows a processor state diagram (similar to the one shown in figure 5-6) for such a system. After initialization, a single goal is selected, and the processor works on it until it finally succeeds or fails. There are then three basic exit states:

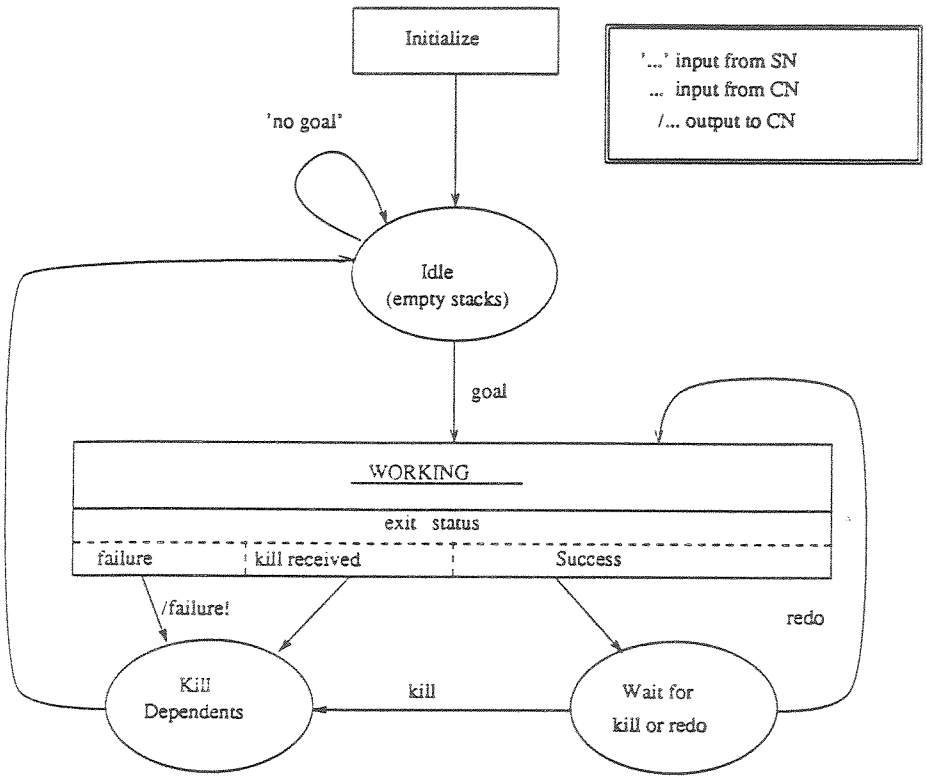

Figure 5-10: State Diagram for the Idle Processor Approach

- In case of **failure** this fact is reported to the parent (i.e. to the processor from whose goal stack the failing goal was "stolen") via a */failure!* message. Also, *kill* messages are sent to all dependents (i.e. all processors which stole goals pushed on to the local goal stack while in the WORKING phase) in order to also discard their parts of the failed computation. Finally the local stack is flushed. Since the local stack is now empty, we are in the same situation as after initialization and we can return to the idle loop to pick up another goal.

- If during the computation a *kill* message is received from the parent (because of failure in an ancestor, as described above) the result is similar to a local failure: all dependents are sent *kill* messages, the local stack is flushed, and a new goal can be picked up.

- In case of **success**, this fact is reported to the parent via a */success* message. Since there will then be structures pending in the local stack, the

processor, instead of picking up a new goal to stack above these structures, simply waits until one of the following messages is received:

- A *kill* message: sent by the parent if an ancestor fails and the part of the computation in this processor has to be discarded. Again descendants are "killed", the local stack flushed, and a new goal can be picked up.

- A *redo* message: sent by the parent if a different solution (alternative) is needed for the same goal. In this case local backtracking is invoked and the processor continues with the new alternative in WORKING mode. If another alternative solution is found, exit will be in success mode. Otherwise it will be in failure mode. Also, a *kill* message may be received during the computation of the alternative.

This scheme avoids stacking problems but it has an obvious and serious drawback: processors are left locked in waiting mode as soon as they produce an answer for the first goal and until backtracking occurs. In such a system, processor utilization would be very low for determinate programs with small granules of computation. Thus, the scheme is obviously only valid for systems with a high number of processors (i.e. those in which the number of processes -available parallelism- is not much higher than the number of processors), and/or which rely on frequent backtracking[39].

[39]Of course, if multiprocessing is available, processors can be kept busy by creating new processes, but then the burden of memory management for the multitude of stacks corresponding to different processes is placed upon the multiprocessing system software and some of the inherent memory efficiency of backtracking systems may be lost. See section 5.3.4.

5.3.3 The Idle Processor Solution - Some Improvement

The Idle Processor Approach can be improved in a simple way by distinguishing between two types of success: success *with alternatives*, and success *without alternatives* (figure 5-11). Note that if a received goal succeeds and at that time it can be determined that there are *no alternative solutions available* (i.e. that the computation was determinate[40]) then the only message that can be received from the parent regarding this goal is a *kill* message. This means that structures for *any* other goal can be stacked above it: if a *kill* message is then received, the section corresponding to that goal is simply deallocated, and we would at most run into the (minor) "garbage slot" problem.

Determinate goals can therefore be stacked until a goal succeeds *with alternatives*. In this case[41], since a *redo* message could arrive, no new goals are stacked on top, thus leaving the top of the stack free for evaluation of another alternative. Of course, if a *kill* message arrives for the goal with alternatives, it is deallocated and execution continues as above.

This scheme would provide much better performance, especially for deterministic cases, for which it is essentially equivalent to the basic model of figure 5-6. Note that this determinate case is more frequent than it may seem because in many systems a "commit" or "cut" construct is provided and it can often be used after the invocation of several goals in parallel, thus eliminating the alternatives in the

---

[40] A simple way of determining this in systems which follow a more or less conventional implementation is by checking whether there is a choice point (or "parcall frame", see next chapter) available.

[41] In a WAM [88] type implementation, there will be at least one choice point in the stack and perhaps some entries in the local trail and heap.

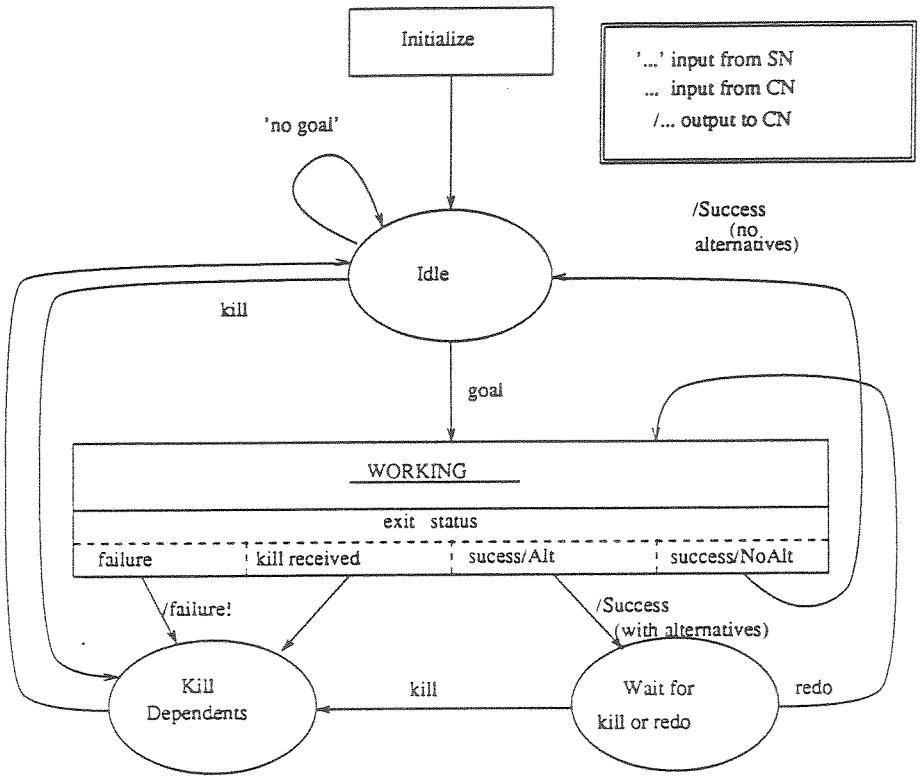

Figure 5-11: State Diagram for the Idle Processor Approach - Improved

scheduled goals[42]. Nevertheless, in this scheme, processors can still idle unnecessarily after evaluating goals with alternatives[43] and complete retrieval of used space on backtracking cannot be guaranteed in all cases.

---

[42]Refer to the *determinate execution algorithms* presented in the previous chapter.

[43]Unless multiprocessing is implemented- see section 5.3.4.

5.3.4 Multi Stack Memory

In conventional systems, processors are a scarce resource when compared to memory. The main problem in the scheme above (i.e. the locking of a processor because of the existence of a goal with alternatives on top of the local stack) is that the presence of such a goal can temporarily preclude the stack from growing. The processor, however, is only waiting, and could continue working on another goal on a different stack. A system where there are more stacks than processors could take advantage of this fact thus greatly improving processor utilization. A processor state diagram for such a system is presented in figure 5-12.

After initialization, a first stack is allocated as usual, and the processor tries to pick up a goal. If one is found, work will start on that goal. In the event of failure or the receipt of a **kill** message, dependents are killed, the stack flushed and a new goal can be picked up using the same stack, as in the approaches previously discussed. But in the event of success, after reporting it to the parent, a *new stack* is allocated so that a new goal can be immediately picked up and worked upon using the new stack. Thus the locking problem is solved. Note that the optimization introduced for the Idle Processor Approach can also be used here: if a goal succeeds without alternatives, there is no need to generate a new stack; the new goal can be stacked above the older determinate goal. The expense of creating new stacks is thus saved for determinate cases.

In this scheme, *kill* and *redo* messages can easily be serviced by identifying the corresponding stack and performing the appropriate operations on them. Note that in a real implementation, outside of our single-stack model, the "creation of a stack" could imply creating a complete new set containing one of each of the areas used in a conventional implementation (i.e. for a WAM type system, a new Stack,

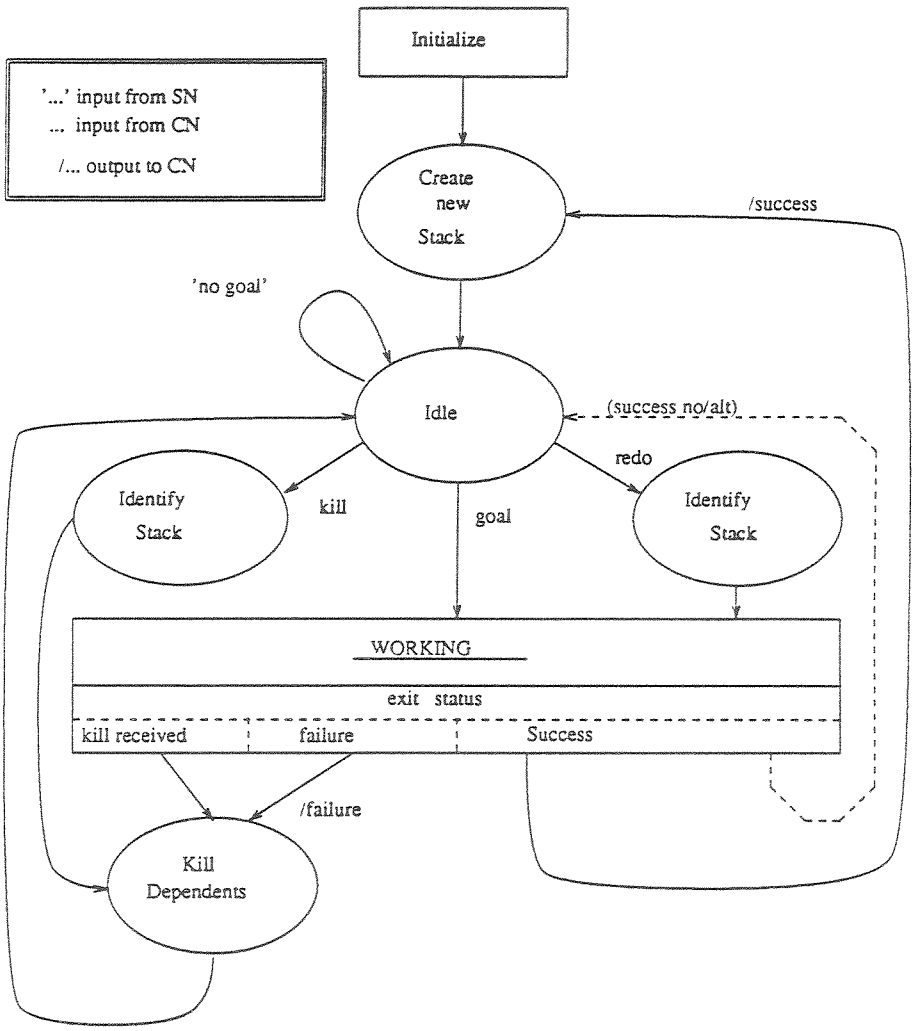

Figure 5-12: State Diagram for a Multi Stack Memory

Heap, Trail, register set etc.). The machine state corresponding to the old stack has to be saved (i.e. the values of all the registers). Really the operation that we have referred to as "creating a new stack" corresponds in more conventional terms to the creation of a new *process*. Thus there will in general be several processes present in a given processor, one for each "stack" in use, some of them "WORKING", some of

them waiting for a *kill* or a *redo*[44]. Processors can then actually be viewed as "workers" which move from one to another of a number of stacks in (shared) memory. Such a multi-stack approach can be straightforward to implement in a system with a large (virtual) addressing space, and virtual memory support. The multiprocessing capability is probably also necessary anyway in any general purpose system where more than one user is to be supported simultaneously. Multiprocessing can also be extremely useful in preventing processors from idling in other cases, which are not directly related to scheduling. One such case is a processor which is waiting for responses from children processors and has no additional local work to do (*join* operation): it can create a new process and new stacks and pick up work from *any* other point in the system[45].

As shown above, the multi-stack model offers very good processor utilization, making it possible to fully support the load balancing characteristics intrinsic to the scheduling algorithm. The obvious additional expense is the need for a more sophisticated memory management system, external to the model, and thus some of the inherent memory efficiency present in backtracking systems may be lost.

---

[44]In an implementation such as the one which will be described in the next chapters, there may also be processes waiting for siblings to report: see figure 5-8-C and -F.

[45]Note that if the goal picked up in these circumstances is a descendant goal, again execution can continue using the same stack and the overhead of creating a new stack can be saved. See the next section.

5.3.5 Goal Restriction

A completely different approach towards ensuring that only *descendant* goals are stacked above other goals is to *restrict the choice of goals which can be picked up by a given processor.* This is in contrast with the schemes presented before, where no limitations were posed on which goals the processor could pick up once it entered the idle loop. The price paid was limited processor utilization and/or poor memory efficiency, or the expense of a more sophisticated memory management system, capable of handling multiple stacks. The advantage was load balancing and an extremely economical scheduling algorithm. In this section we will reconsider the scheduling algorithm instead.

Figure 5-13 shows the processor state diagram for a parallel backtracking system with *distributed goal scheduling restriction.* After initialization a processor can always pick up any goal in the system (its stacks are empty, so there is no need to take into account any underlying structures). In general, work on any goal will finish in one of the previously discussed circumstances (exit status). The most interesting difference occurs when this status is *success with alternatives.* As before, this means that the goal has been executed successfully, and there are still more possible solutions (i.e. at least one "choice point") for it, the stacks thus containing pending structures. In such circumstances, the *goal restriction approach* dictates entering an *alternate idle loop*, where, rather than picking up any available goal in the system, only an *appropriate* goal (defined as one which is a descendant of the last goal received) will be looked for. Only when such a goal is found, work will continue on it, and in confidence that its data structures can safely be grown on top of the old ones, since the descendence relationship ensures that the newer structures will always be deallocated before an underlying goal needs to be backtracked.

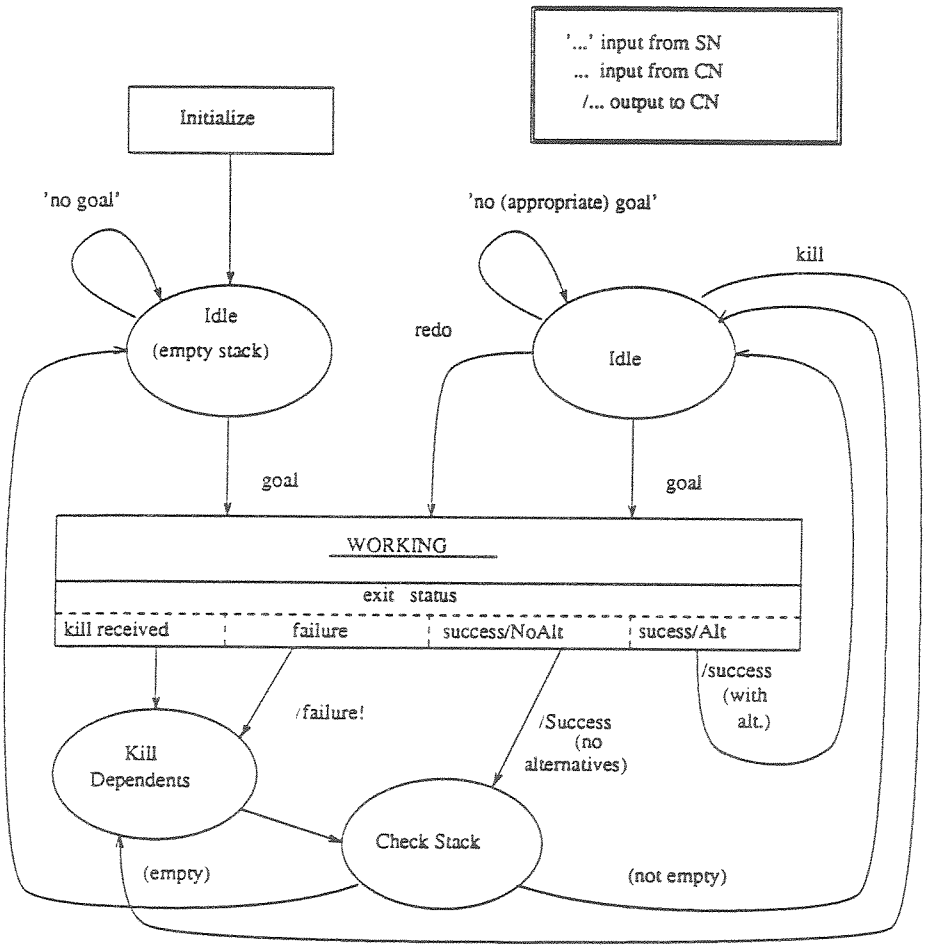

Figure 5-13: State Diagram for the Goal Restriction Approach

If the goal exits in any of the other possible states (*failure*, *kill received*, or *success with no alternatives*) then, after the corresponding actions are performed (reporting to parent, killing dependents etc.), the local stack is checked to see if there are any pending structures in it. If the stack is empty, then a loop looking for *any* available goal is entered. If it is not empty, then the loop looking for an *appropriate* goal is used.

One important advantage of this method is that, since thanks to the selection of goals the ordering of goals in the stack corresponds to its backtracking order, *any* **kill** *or* **redo** *message received from the parent necessarily refers to the* **last** *goal received.* This obviously means that all allocation and deallocation of structures on all stacks is done always to and from their tops. It also means that *kill* and *redo* messages do not have to identify the goal they refer to: it is always the last goal executed in the processor.

Of course, some method has to be devised for determining descendance relationships between goals. One solution which can be used has been proposed in [55] (applied to determining variable "age" in an OR-Parallel system): a simple "block number" determines the relative *age* of different "chunks" of memory. Ait-Kaci, Boyer, and Nasr have proposed an encoding method (applied to the determination of the Greatest Lower Bound in a novel unification algorithm for supporting "inheritance" [35]) which makes it possible to determine these relationships by performing a low overhead boolean check. It is conceivable that this function could be supported in hardware through a "Descendant" network, similar to the scheduling network in figure 5-5. A similar scheduling algorithm to that shown in section 5.2.3 could then be used, in which a processor would feed in the level of the goal on top of its local stack to the "Descendant" network, and the network in turn would return a Pid. for the processor having the highest level goal which is a descendant of the requested level in its goal stack, i.e. ready for execution. However, since this encoding technique is presented in [35] as a compilation (i.e. static) technique, and it is currently valid for partial orderings only, its applicability (or modification) for this purpose remains an area open for research.

Other solutions can also be applied. For example, the processor can take

goals from any random processor (as given by the scheduling network) when the stack is empty, and only from the goal stack of a parent processor (i.e. the one from which the top goal in the stack was taken) when there are structures pending. This policy also guarantees correct goal ordering without labeling goal levels.

5.3.6 A Combined Approach

The solutions proposed in the last two sections have complementary characteristics: the multi-stack solution effectively solves the problem of having inactive processors for all cases, but runs into the additional overhead involved in the creation and deletion of multiple stacks. The goal restriction approach makes much more efficient use of its stack, but it can sometimes leave processors idle, even though there is work available in the system, just because the available goals are not "appropriate".

A combined approach, such as that shown in figure 5-14, can offer the advantages of both models. Here, an idle processor will first look for *appropriate* goals in the system. As long as *appropriate* goals can be found, they are executed on the existing stack taking advantage of the inherent memory management efficiency of the single stack approach. If at any point no *appropriate* goals can be found, but there is work available in the system (i.e. *non-appropriate* goals) then a new stack is created and work continued on it. Note how this eliminates the occurrence of idle processors if there is work available in the system while at the same time taking advantage of the inherent efficiency of the single stack model whenever possible. Also note how as the number of stacks grows, the scope of "appropriateness" of goals expands too: now *all descendants of all the goals on top of all stacks* are *appropriate*. Thus, as the occurrence of non-appropriateness decays, the number of stacks generated should be bound.

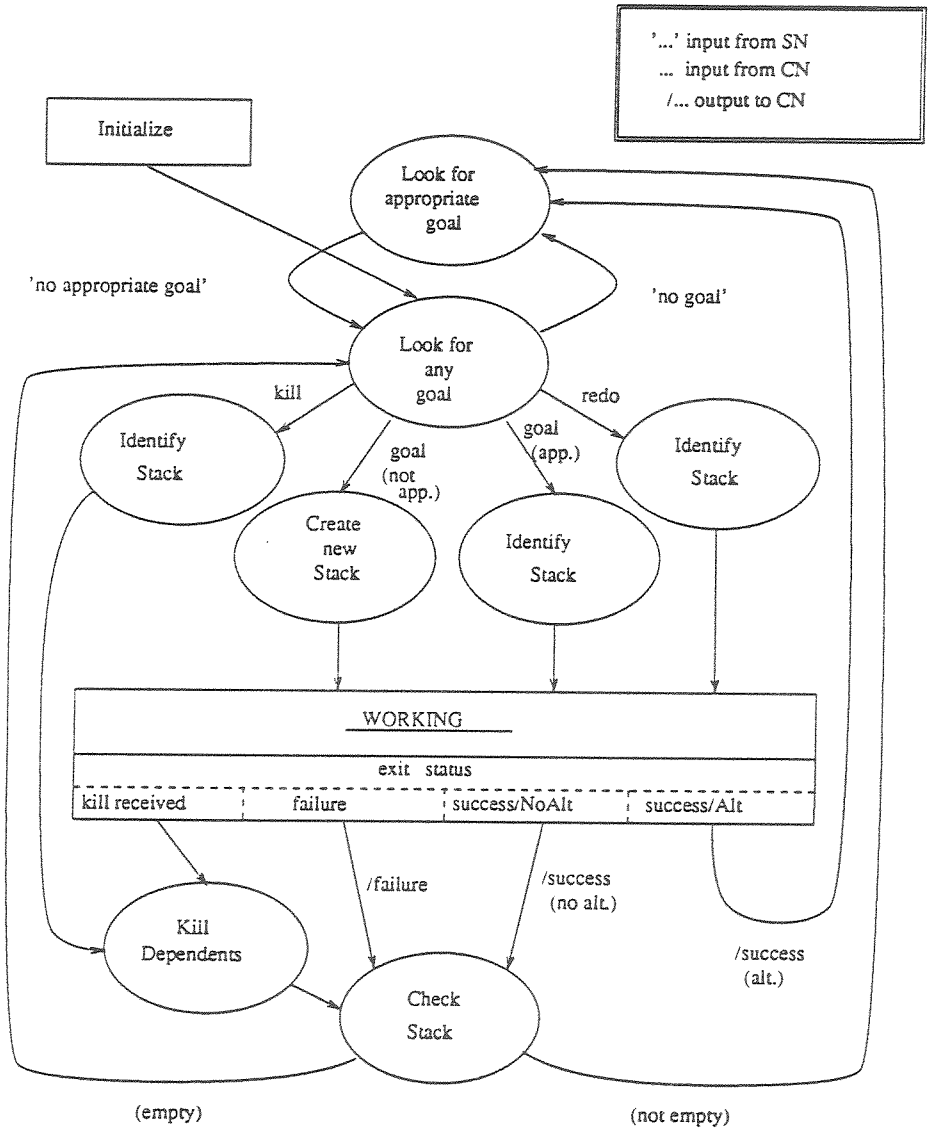

Figure 5-14: State Diagram for the Combined Approach

5.4 Chapter Summary

In the previous sections we have discussed the interactions between two important issues in stack-based parallel logic program implementation: scheduling and memory management. Single and multi-stack memory organizations, and restricted and unrestricted distributed goal scheduling strategies were proposed and analyzed. Special emphasis was placed on their application in backtracking systems, studying their special features and problems. A combined multi-stack/goal restriction model showed the best performance potential and is indicated for larger granularity systems where virtual memory and scheduling hardware support is available. This is the case in most current multiprocessor systems, where a limited number of relatively large processors are interconnected and limitation of processor idle time is essential. Such a combined approach eliminates idle processors if there is work available in the system while taking advantage of the memory efficiency of the sequential model whenever possible. This is obtained at the expense of a more sophisticated memory management system, capable of handling multiple stacks. At the other end of the spectrum, in systems containing a very large number of small processors, simpler solutions which entail lower overhead and, therefore, potential for better performance, can be applied at the expense of lower processor utilization. If some appropriate scheduling hardware is available, then the goal restriction approach can offer good processor utilization and excellent memory efficiency in a medium size granularity system.

In the next chapters an implementation model for Parallel execution of Logic Programs will be presented which is based on the goal scheduling and memory management schemes discussed in this chapter, and which implements the procedural semantics developed for Horn Clauses annotated with Conditional Graph Expressions in the previous chapter. This scheme will be described down to the abstract machine

level in order to establish the feasibility of the implementation and make realistic simulations possible. The following chapter will describe a series of techniques which can be used at the Abstract Machine level for supporting the algorithms of chapters 4 and 5. Chapter 7 will then present the data areas and instruction set of an Abstract Machine design which makes use of such techniques.

Chapter 6

Implementing Distributed Backtracking at the Abstract Machine Level

This chapter introduces some of the techniques which will be used in the following chapter for the implementation of distributed backtracking at the abstract machine level. The purpose of these techniques is to support the algorithms which were introduced in Chapter 4 for forward and backward execution of Horn Clauses annotated with Conditional Graph Expressions with minimum overhead and in a manner compatible with the goal scheduling and memory management strategies described in the previous chapter.

A main design objective is to make these techniques *compatible* with those used in high performance sequential implementations. This objective is based on the fact pointed out in previous chapters that, although logic programs can present considerable opportunities for AND-Parallelism, there are always (determinate) code segments requiring sequential execution. A system which can support parallelism while still incorporating the performance optimizations and storage efficiency of current sequential systems is thus desirable. The approach taken herein of providing the mechanism for supporting forward and backward execution models for AND-Parallelism as extensions to the mechanisms used in a high performance sequential implementation offers two major advantages: first, sequential execution remains as *fast* and *space efficient* (thus avoiding garbage collection as much as possible) as in the high performance sequential implementation (modulo some minimal run-time

checks); second, the model is offered in the form of *extensions*, which are fairly independent, in spirit, of the peculiarities of that implementation. Therefore, the approach described here is applicable to a variety of compilation/stack based sequential models.

6.1 Implementing Sequential Logic at the Abstract Machine Level: The WAM

The basic storage retrieval mechanisms used during backtracking in current sequential systems were already described in the previous chapter. However, the highly simplified model offered therein left out many issues which are relevant in practice. Before the strategies for implementing **CGE** based AND-Parallelism with the associated backward execution mechanism are presented, a more accurate description of sequential backtracking implementation techniques will be given in this section. This description will be based on one of the highest performance Prolog implementations to date: the Warren Abstract Machine (**WAM**) [88]. The **WAM** will not only constitute the starting point for the description of the distributed backtracking techniques in this chapter, but also the basis for the *parallel abstract machine level design* of the next chapter.

6.1.1 Data Areas and General Operation of the WAM

The **WAM** [88] is an efficient execution model coupled with a host of compilation techniques leading to one of the fastest and most efficient implementations of Prolog today. The ideas it incorporates are believed to be a major breakthrough in the design of computational logic systems [55]. Lack of space prevents us from fully describing the **WAM** here, but we will point out those basic concepts which are necessary for understanding the discussion in this and the next chapters. For a complete description of the **WAM** the reader is referred to Warren's original SRI report [88] or to the tutorial on the **WAM** available from Argonne Labs [29].

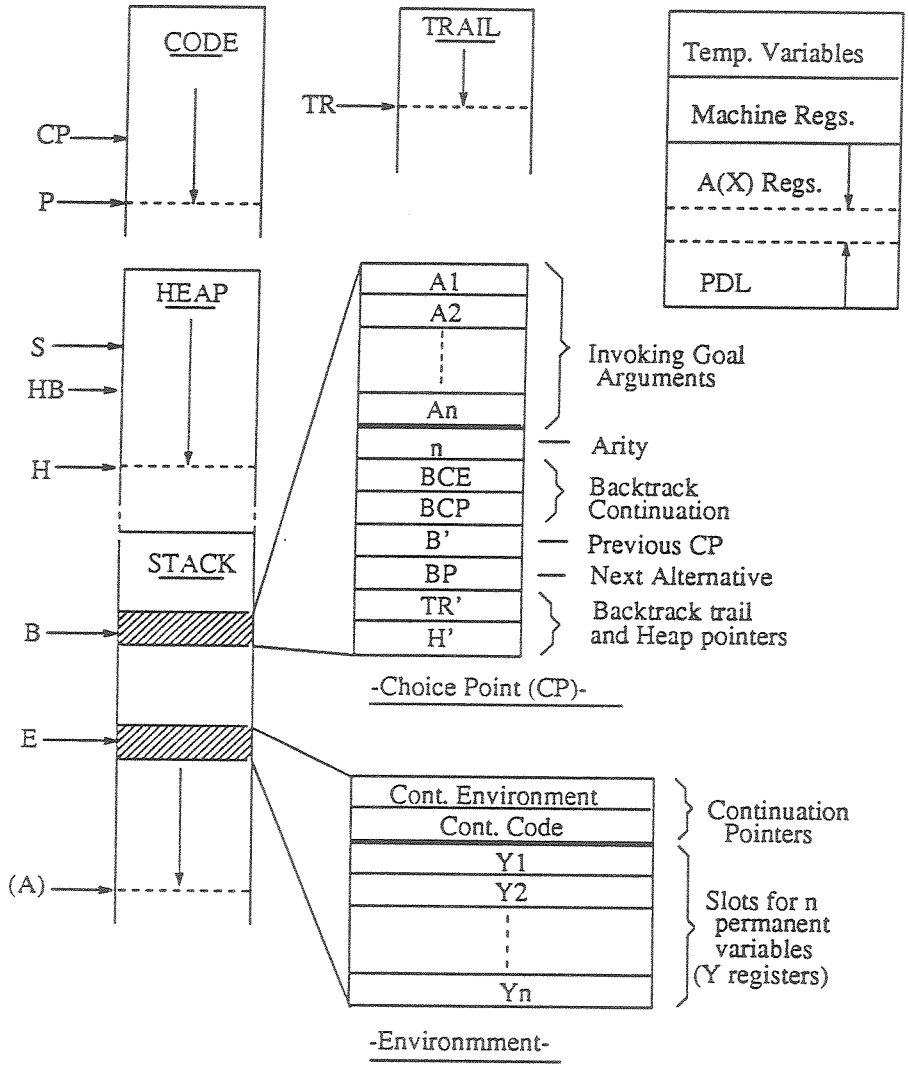

Figure 6-1: Data areas and registers for the WAM

Figure 6-1 shows a general view of the data areas of the WAM. They include:

- The *Code* area: which contains the program *in compiled form*. The next instruction to be executed is pointed to by register P.

- The *Heap*: where data structures and long-lived global variables are

created, updated, and discarded (upon backtracking). Structure copying [47] (rather than structure sharing) is used in the *Heap*: new structures are pushed on to the *Heap* explicitly, as modified copies of old ones. Register **H** points to the top of the *Heap*.

- The *Stack*: which contains two types of objects: *environments* and *choice points*.

    - An *environment* contains a number of value cells which are used to store (permanent) variables which can be accessed by the goals within the body of the clause or by children clauses called by these goals. It also contains some continuation information which is equivalent to the return address in a subroutine call: it points to the instruction in the body of the calling clause where execution will continue after the called clause finally succeeds. Register **E** points to the current *environment*. An *environment* is pushed on to the *Stack* every time a clause with "permanent variables" is entered. Environments which are no longer needed (for example before the last call in a clause) can be discarded ("last call optimization" [87]).

    - A *choice point* is pushed on to the *Stack* when the first clause of a set of alternative clauses is entered. It contains all necessary information to restore the state of the machine and a pointer to the next alternative clause. Upon failure, backtracking is accomplished by finding the last *choice point* in the *Stack* (pointed to by register **B**), reloading all machine registers from its contents, and restarting execution at the alternative clause. This will be explained in more detail in the next sections. Resetting the registers takes care of discarding the top of the *Heap* and *Stack* (i.e. discarding variables and structures created since the *choice point*). However, some *variable instantiations* may have been made deeper in the data areas which need to be undone upon backtracking. This is taken care of by

- The *Trail*: where variable instantiations which need to be undone are recorded (one entry for each variable). These entries are used on backtracking to restore the corresponding variables to "uninstantiated". This operation is called "detrailing" or "unwinding" the *Trail*. Register **TR** points to the top of the *Trail*.

In addition to the data areas (*Code/Stack/Heap/Trail*) there are other elements in the design of the **WAM**: a number of argument registers (called A or X

registers) are used for passing arguments when a procedure (i.e. a collection of clauses with the same head functor and number of arguments) is called. There is also a small "Push-Down List" (*PDL*) which is used by the recursive general purpose unification routine as a call stack.

Prolog programs are *compiled* into a series of abstract machine level instructions which perform different operations on the above mentioned areas. In order to broadly describe the function of some of these instructions, a normal procedure call ("goal invocation") sequence will be followed: the first step involves loading the argument registers (A1 through An, where n is the number of arguments in the call -the Arity of the procedure) with the appropriate values; *"put"* instructions are used for this purpose. The procedure is then called (*"call/execute"* instructions). Upon entry into a procedure, a *choice point* is created if it has more than one alternative (*"try"* instructions) and then each of the terms in the head of the clause is unified (*"get/unify"* instructions) with the corresponding argument loaded in (or pointed to by) the argument register. If unification does not succeed, failure occurs and backtracking to the last *choice point* will occur. *"Get"* instructions are used to encode at compile-time cases where unification defaults to a simple assignment or a set of very simple determinate steps. Because the main activity of a Prolog program is centered around unification of goals with candidate clauses, the simplification of this step results in important performance improvements.

The WAM offers many other features designed towards improving speed and space economy, such as retrieval of all used space upon backtracking, last call optimization, and *environment* trimming. Instructions are also provided for supporting the technique of indexing the clauses based on the first argument. This reduces the number of alternatives to be tried and has an important role in improving execution speed and detecting determinate cases.

6.1.2 Backtracking in the WAM Revisited

Backtracking is one of the basic operations that the **WAM** is designed to support efficiently. Since the **WAM** backtracking mechanisms constitute the starting point in the following description of distributed backtracking, they are worth looking at in more detail. This is best done through an example.

Figure 6-2 corresponds to the execution of the following group of clauses (labels have been given to the different clauses involved in order to name the different alternatives within each procedure):

```
procedure a:
a1:   a :- b, c, d, e.
a2:   a :- b, c, d, e.
a3:   a :- b, c, d, e.

procedure b:
b1:   b :- ..., ..., ... .
b2:   b :- ..., ..., ... .
b3:   b :- ..., ..., ... .

procedure c:
c:    c :- ..., ..., ... .

procedure d:
d:    d :- ..., ..., ... .

procedure e:
e1:   e :- ..., ..., ... .
e2:   e :- ..., ..., ... .
e3:   e :- ..., ..., ... .
```

For simplicity, only the *Stack*, the *Heap*, and the *Trail* are represented in figure 6-2. Observe in this figure how, upon entering *procedure a:*, since **a** has alternatives, the corresponding *choice point* is created in the *Stack*. Execution of **a** then starts with the first alternative *a1:*. This situation is depicted in figure 6-2-A. Only the following information included in the *choice point* is shown (other information will be skipped for the sake of brevity):

- A pointer to the next unexplored alternative clause *a2:*.
- The value of the *Heap* pointer in register **H** at the time this *choice point* was created[46] .

[46] A further optimization is actually implemented in the **WAM**: the *previous* value of **H** (saved in register **HB**) is actually stored. This simplifies the decision of when to trail variables.

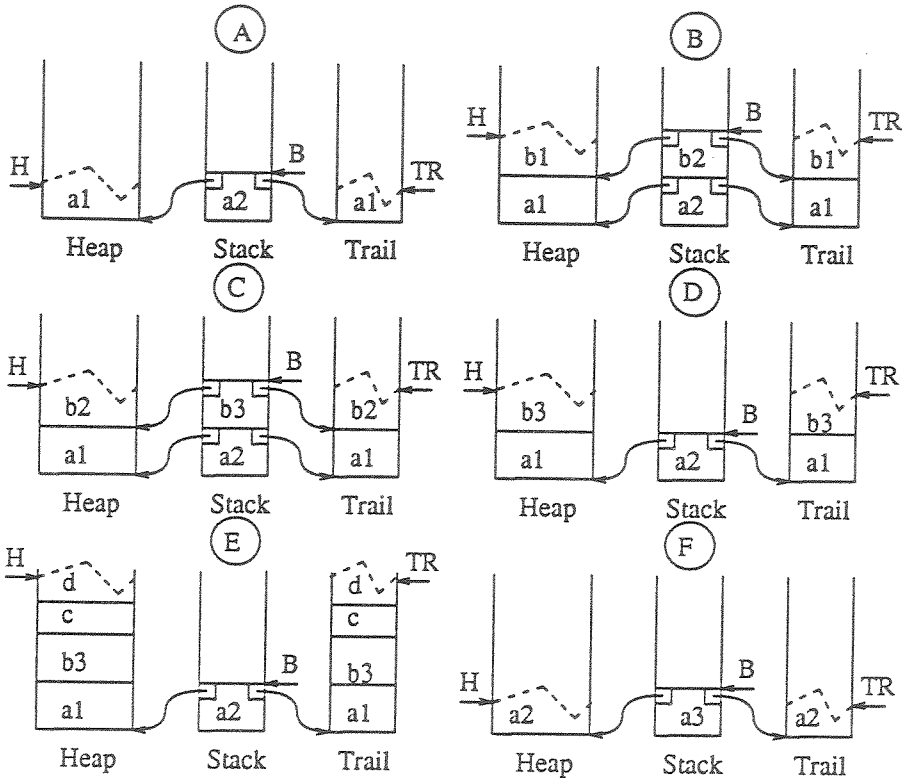

Figure 6-2: Choice Point Based Backtracking in Sequential Systems

- The value of the *Trail* pointer in register **TR** at the time this *choice point* was created.

When the head of *a1:* unifies successfully with the invoking goal, *procedure b:* is entered. Again a *choice point* is created, since **b** also has alternatives (figure 6-2-B). Suppose now that some goal fails in the body of *b1:*, and that no more *choice points* have been created. The following sequence of actions takes place resulting in *backward execution* (this is illustrated in figure 6-2-C):

- The most recent *choice point* is fetched through register **B**'s content.

- The top of the *Heap* pointer (register **H**) is reset to the value saved in the fetched *choice point*. This will discard all the data just made obsolete by the failure that caused the backtracking. As mentioned before, Prolog relies heavily on this retrieval of space during backtracking in order to avoid garbage collection.

- The variables remembered through entries located between the current top of the *Trail* stack and the *Trail* pointer saved in the fetched *choice point* are reset to uninstantiated. This is done because the instantiations being reset were made obsolete by the failure that caused the backtracking. Of course the top of the *Trail* pointer (register **TR**) is also reset appropriately.

- Finally, the next alternative *b2:* indicated in the *choice point* is picked up and execution proceeds from there. The fact that the next alternative clause is *b3:* is recorded by updating the *choice point* appropriately.

If **b** should fail again, the above sequence of actions would be repeated, and execution of *b3:* started. However, this time there would be no more alternatives for *procedure b:*. This means that the *choice point* associated with *procedure b:* should be discarded and register **B** should be reset to the most recent one prior to the one being discarded. This is only possible if the *choice points* are chained together (This is one of the information items that are not shown in the *choice point* frames illustrated in figure 6-2).

In figure 6-2-E the situation is depicted after *b3:* and *c:* have succeeded, and *d:* is being executed. Note that since neither *c:* nor *d:* have alternatives, no more *choice points* have been created on the *Stack*. Therefore, if *d:* should fail at this point, the general backward execution model using the current most recent *choice point* (fetched through register **B**) would correctly lead to alternative clause *a2:*. This is shown in figure 6-2-F. Some interesting points to be noted are:

- This implementation achieves efficient garbage collection of *Heap* space upon backtracking: all data created there during forward execution are discarded automatically by appropriately resetting register **H**.

- Identifying the most recent *choice point* is immediate, since it is always pointed to by register **B**.

- *Choice points* are only created when they are needed (i.e., when the clauses

have alternatives) and they are discarded efficiently when they are not needed any more.

6.2 Implementing Distributed Backtracking in AND-Parallel Systems

As stated before, the objective in this chapter is to develop techniques in order to support forward and backward execution of Horn Clause programs annotated with **CGE**'s as an extension of those described in the previous section for the **WAM**. The basic forward and backward execution algorithms were already introduced in Chapter 4. The idea in this section is to support those algorithms while still preserving the efficiency present in sequential implementations. As mentioned in that chapter, "Point Backtracking" will be assumed for the rest of this discussion.

The conceptual starting point is that described in the previous chapter: execution starts at a given processor, and it continues sequentially until a **CGE** is encountered whose conditions evaluate to true. Since at this point all the goals inside the **CGE** (a collection of "AND siblings") can be executed in parallel, the processor which is running the **CGE** (the "parent" processor) pushes these goals on to its *goal stack* and they are picked up from there by other processors which will be in charge of executing each of these goals. As seen in the previous chapter, each of these processors will have its own execution environment (*Stack*, *Heap*, *Trail*, as well as a machine state). Of course, one of the natural extensions to such a general model is to let the parent processor execute one or more of the goals inside the **CGE** instead of just idling while waiting for other children processors' responses: this will be discussed in more detail in section 6.3 on local execution of parallel goals, showing how the existing data areas (*Stack*, *Heap*, and *Trail*, etc.) can be shared for this purpose.

In this model, then, the parent will be in charge of the "fork" and "join"

operations needed by the forward semantics of the **CGE** (i.e. making the goals inside the **CGE** available for parallel execution and waiting for their completion before continuing beyond the **CGE**). It is also in charge of supervising the generation of alternatives as dictated by the backward execution algorithm. The control structure that the parent uses for its supervisory task will be referred to as a "parallel call" frame (*Parcall frame* in short) and will be located in the parent's *Stack* (therefore three types of frames can now be found there: *environments*, *choice points*, and now, *Parcall Frames*). The most recent *Parcall Frame* is pointed to by register **PF**. *Parcall Frames* are created when a **CGE** evaluates to **true**, hence clearing the way for the parallel execution of the **CGE**'s sibling predicates. The *Parcall Frame*, among other information, contains the following items important for this discussion[47]:

- One slot for each of the AND-Parallel procedure calls inside the **CGE**, consisting of the following fields:
  - the Id of the child process corresponding to this procedure call
  - completion status of the process (i.e. *processing*, *succeeded with pending alternatives*, *succeeded with no alternatives*, or *failed*).
- A flag indicating whether the **CGE** has just been entered or whether it is being backtracked into after the initial entry and at least one successful exit. This is a materialization of the "inside"/"outside" indication discussed in the backtracking algorithm of Chapter 4.
- The current values of the pointers (registers) into the data areas (this part is also referred to as the *"Wait Marker"*).

In the next paragraphs it will be shown how the introduction of *Parcall Frames*, their relationship to *choice points*, and the manipulation of both types of frames will materialize the algorithms introduced in Chapter 4 and make it possible to

[47] Other information is also needed in practice which is not relevant to this discussion. These details will be completed in the next chapter.

manage both forward and backward execution as a natural extension to the WAM model. First, two types of failure are defined:

- *Local Failure*: the local processor fails while executing a goal, and
- *Remote Failure*: a "Failure" message is received from a child process.

Now the extended backward execution mechanism is based on recognizing, when either type of failure occurs, whether a *choice point* or a *Parcall Frame* is more recent (comparing registers **B** and **PF**). The algorithm then follows:

- If *Local Failure*, then:
  - If **B** > **PF** then perform the normal *choice point* backtracking.
  - If **PF** > **B** then find the first[48] *Parcall Frame* child process slot with pending alternatives to respond successfully to a "redo" message ("unwind" messages are sent to all previous slots). When such a process is found, invoke the parallel execution of all the goals which correspond to the following slots, thus returning to (parallel) forward execution. If none succeeds, fail by recursively performing this backward execution algorithm in a "local failure" mode.
- If *Remote Failure*, then, knowing definitely that **PF** > **B** and that it is the "inside backtracking" case (that is until the "local goals" optimization of the next section is introduced):
  - "Kill" all goals in the *Parcall Frame*, fail by recursively performing this backward execution algorithm in a "local failure" mode ("restricted" intelligent backtracking).

The following example will illustrate the above algorithm. Suppose the clauses for "a" in the example in the previous section were annotated in the following way (with embedded **CGE**'s):

---

[48]Slots should always be scanned in the same order, e.g. from the higher addressed ones (hopefully corresponding to rightmost ones in the **CGE**) to the lower addressed ones.

```
procedure a:
a1:   a :- ( cond1 | b & c & d ), e.

a2:   a :- ( cond2 | b & c & d ), e.

a3:   a :- ( cond3 | b & c & d ), e.
```

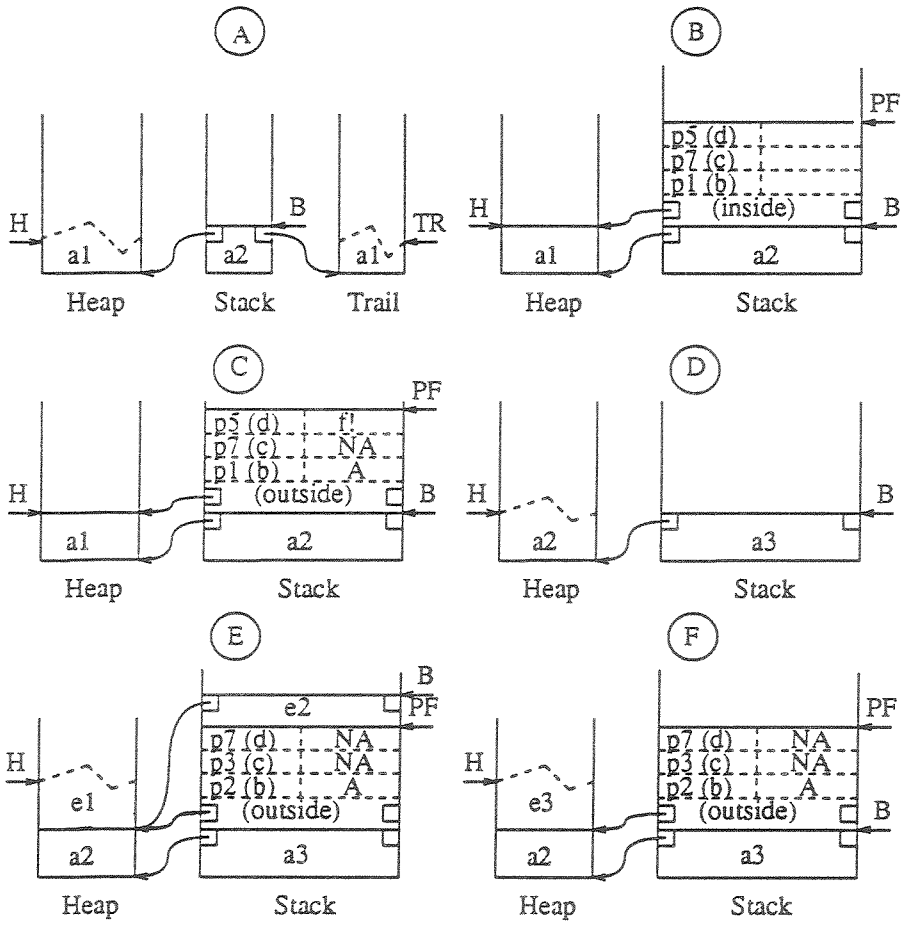

Figure 6-3: CP/*Parcall Frame* Based Backtracking in AND-Parallel Systems

Figure 6-3 illustrates the execution of this example in parallel. Execution of **a** in the "parent" process starts exactly as in the sequential case (figure 6-2-A vs. figure 6-3-A). If **cond1** failed, execution would proceed just as in figure 6-2. On the

other hand, if **cond1** succeeds, a *Parcall Frame*, initialized to "inside" is created, with slots for **b**, **c**, and **d**. This is illustrated in figure 6-3-B where these goals have been "picked up" by **p1**, **p7**, and **p5** respectively (the *Trail* is omitted in both the diagrams and the discussions for the sake of clarity). At this point the parent process simply waits for all goals to return (flagged by the updating of their slot's *completion status* field). With the *Parcall Frame* still flagged as "inside", if one of the goals returns failure (*Remote Failure*) execution can backtrack "intelligently" to the last *choice point* before the *Parcall Frame*. In figure 6-3-C, **p5** returned failure for **d** (**p1** and **p7** returned with success, with **p1**'s success qualified as with pending alternatives, i.e. there is a *choice point* in **p1**'s *Stack*). Since the corresponding *Parcall Frame* is still flagged as "inside", an "unwind"[49] message is sent to **p7** and **p1** (thus disregarding the alternatives in **b**), and execution is continued with the next alternative of **a** (figure 6-3-D).

The next two parts of figure 6-3 illustrate "outside" backtracking. Figure 6-3-E depicts a situation similar to that in figure 6-3-B. Processors **p2**, **p3**, and **p7** "picked up" the goals but this time they all returned successfully (b still having alternatives). At this point the whole **CGE** succeeds by changing the status of the *Parcall Frame* to "outside", and execution moves on to goal **e**, pushing a *choice point* (since **e** has alternatives), and finally entering clause *e1:*. If *e1:* fails, the available *choice point* will be used to try *e2:* (*Local Failure*; **B > PF**). Figure 6-3-F illustrates the situation if *e2:* also fails: the *choice point* has been deallocated and *e3:* is now being executed.

---

[49] "Unwind" messages instruct the processor which receives them to free all storage corresponding to a particular goal and to unwind the corresponding portion of the *Trail*. Note that if the techniques presented in the previous Chapter are used, this retrieval will always be complete and from the top of the corresponding stacks. "Kill" messages serve the same purpose, but they are used *when the corresponding process is still running*. Therefore they imply aborting execution before the storage retrieval is performed.

Note that in the event of a *local* failure now, the last *Parcall Frame* is more recent than the last *choice point* (**PF** $>$ **B**) and, since its status is "outside", the corresponding backtracking algorithm will be run on it: select the first goal with alternatives (**b**), send a "redo" to it (to **p2**, which will execute it by making use of the *choice point* on top of its local *Stack*, just as if a local failure had occurred) and "unwind" messages to the ones to its right (i.e. the previous slots, p3/**c** and P5/**d** in this case) so that their *Heaps* will be deallocated and their *Trails* unwound. If **p2** now returns failure, since there are no more slots with alternatives in the *Parcall Frame*, it will be deallocated and the next entry on the *Stack* (**a**'s *choice point*) will be used to backtrack to *a3:*. If, on the other hand, **p2** had returned success, the parallel execution of all the goals corresponding to the following slots will be reinvoked (**c** and **d**), hence "shifting gears" to "Forward Execution". Note that it can be safely assumed that the **CGE** will be successfully exited at this point since those goals are being redone from scratch and it is known that they have succeeded in the past!

6.3 Local Execution of Parallel Goals

One obvious optimization to the scheme above is to let the local processor pick up some of the goals in the *Parcall Frame* and work on them itself, instead of just idling while waiting for children processes' responses. This is very important in that it allows the generalization of the architecture to any number of processors (including a single one). Such scalable systems could then run parallel code with "graceful" performance improvement or degradation depending on the available resources. Also, a single processor would run the parallel code at comparable speed to equivalent sequential code, while still taking advantage of the opportunity for "intelligent backtracking" present in "inside" backtracking.

6.3.1 "Local Goals First" (LGF) Backtracking

In a multiprocessing system, local execution of parallel goals can be accomplished by creating a new process locally which will pick up one of the goals in the local goal stack. However, in the last chapter it was pointed out how when a goal is known to be a descendant of the last goal executed by a processor, it can be safely picked up and executed in the knowledge that it will be backtracked and/or deallocated before any of the underlying structures. A "local goal" is by definition a descendant of any other goals in a processor's stacks. Based on this observation, figure 6-4 shows a more efficient way of handling the execution of parallel goals locally, by stacking them on the local data areas much in the same way as they would be in a sequential implementation. In figure 6-4-A, *b1:* has been immediately "picked up" by the local processor (and the corresponding slot has been marked accordingly -- "=*=") while **c** and **d** have been "picked up" by **p7** and **p5**, as in figure 6-3-B. Execution of the goal taken locally proceeds as normal (figure 6-4-B), but note that the *Parcall Frame* is still marked as "inside". In this figure **p5** has returned (with *no alternatives*) and **p7** is still working on its goal. In the event of either a local or a remote failure now, "inside" (i.e. "intelligent") backtracking would occur (as in figure 6-3-D). For example, this would be triggered locally if **b** runs out of alternatives. A first failure in *b1:* in figure 6-4-B, however, would simply use the *choice point* and continue with *b2:*, just as if it were being executed remotely.

If all goals succeed, execution will continue with **e**, data structures and *choice points* being again simply pushed on top of their respective areas (*Heap* and *Stack*, figure 6-4-C). "Outside" backtracking will work in a similar way as before, but with the difference that *goals executed locally will always be backtracked first*: in figure 6-4-C, if **e** runs out of alternatives, all the alternatives of **b** will be tried before using the *Parcall Frame*. This is perfectly valid, as long as it is used

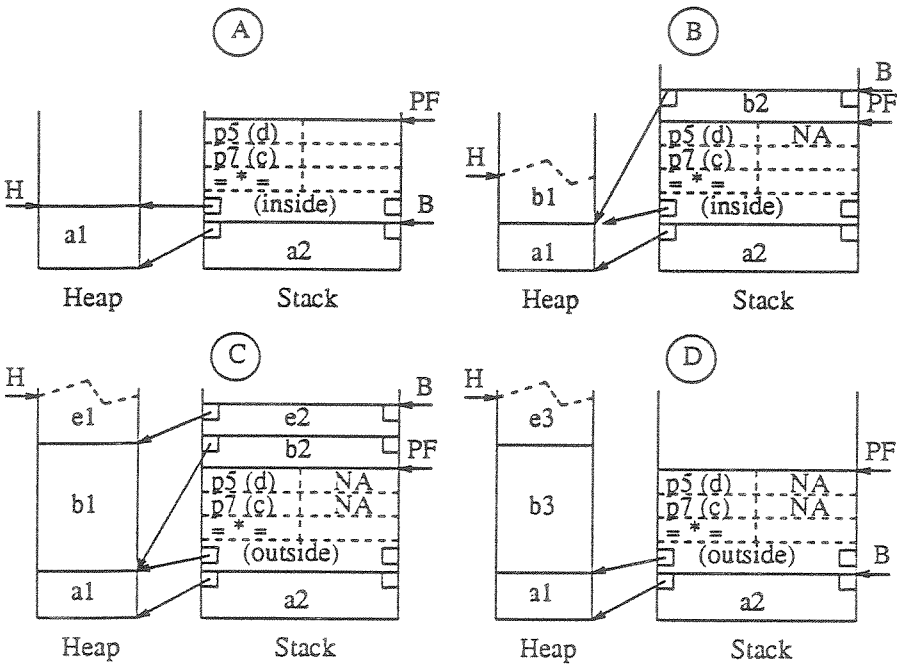

Figure 6-4: "Local Goals First" (LGF) Backtracking

consistently, since the order of execution is immaterial inside a parallel call. The *Stack* status of figure 6-4-C is therefore equivalent to the one which corresponds to the execution of the following clause using the scheme described in the previous section:

```
a :- ( c & d ), b, e.
```

This method of supporting execution of local goals is therefore termed "Local Goals First" (**LGF**) backtracking.

Figure 6-4-D depicts "outside" backtracking after all goals executed locally have run out of alternatives. *e3:* is being executed after completion of *b3:* (both *choice points* have been discarded). If failure occurs at this point in *e3:*, the *Parcall Frame* will be found above any *choice points*, and the "outside" algorithm will be executed on it. In this case, since no goals in the *Parcall Frame* have alternatives, the *Parcall Frame* itself will be discarded (sending "unwind" messages to **p5** and **p7**) and the next alternative of **a** (*a2:*) will be tried next as in figure 6-3-D.

An interesting situation occurs if external failure arrives while the local processor is executing a goal from the parallel call, and this goal in turn has generated other *Parcall Frames*. Suppose that in figure 6-4-B execution of *b1:* has pushed other *choice points* and *Parcall Frames* on the *Stack*. If **p7** (**c**) returns at this point with failure, all those entries, and their corresponding data structures (in the *Heap*) have to be deallocated. This turns out to be simple if **p7** provides the value of the PF pointer for the *Parcall Frame* containing the goal failing (it can be "picked up" with the goal). This frame is referred to as the "failing *Parcall Frame*". Then the backtracking information contained in that Frame is used to recover all space (i.e. just above *a1:* for the *Heap* in figure 6-4-B). Of course, all processes started by the execution of **b** need to be cancelled. This is accomplished by following the chain of *Parcall Frames*, from the one on top to the one given by **p7**, sending "kill", "unwind" etc. messages to all slots that are not marked local ("=*="). This is very similar to what a processor has to do when it *receives* a "kill" message.

In summary, an algorithm along the same lines as the one presented in the previous section can be used when **CGE** goals are executed locally, provided it is adapted to handle the extra special cases involved:

- If *Local Failure*, then:
  - If **B** > **PF** then perform the normal *choice point* backtracking.
  - If **PF** > **B** and the status of the *Parcall Frame* is "inside", "kill" all goals in the *Parcall Frame* (by sending "kill"/"unwind" messages to all non-local slots in this *Frame*; local goals will be deallocated automatically by the local trimming of the stacks) and fail by recursively executing this algorithm in a *Local Failure* mode.
  - If **PF** > **B** and the status of the *Parcall Frame* is "outside",

then find the first[50] *Parcall Frame* child process slot with pending alternatives to respond successfully to a "redo" message (sending "unwind" messages to previous slots). When such a process is found, invoke the parallel execution of all the procedure goals that correspond to the following slots, and of all those calls which were executed locally[51] . If none succeeds, fail by recursively executing this algorithm in a *Local Failure* mode.

- If there are no *choice points* or *Parcall Frames* available, report failure to parent.

- If *remote failure*, then:
  - If the **PF** value received is the same as the current one: this case is equivalent to the second situation above.
  - If the **PF** value received is lower than the current one: follow chain of *Parcall Frames* "killing" dependent processes up to and including referred Frame; fail by recursively executing this algorithm in a *Local Failure* mode.

Note that although the description is lengthy because of the different cases involved, the abstract machine can select the appropriate case using simple arithmetic checks (**B** $>$ **PF** or **B** $<$ **PF**; Status$=$ 1 or 0) and the actions are in any case very simple and determinate. Backward execution can be performed in parallel (i.e. unwinding of *Trails*, killing of descendants, etc.) with very little overhead. Then forward execution is resumed also in parallel.

---

[50]The correct scanning order now is *opposite to that in which the goals were picked up by remote processors*. A simple way of following this order is by making use of an extra field in the child process slot which stores the "outgoing order" of local goals.

[51]Note that all local goals have been completely backtracked before this point is arrived at.

6.3.2 "Right Goals First" (RGF) Backtracking

In the **LGF** model described above the order of execution during backtracking differs from that of a sequential implementation: *local goals are backtracked first.* Since there is no a priory knowledge of which goals will be executed locally, the *order* in which solutions are produced depends on run-time factors, even though *all* solutions will still be produced. Such an approach, although offering the advantage of a very efficient implementation, has several drawbacks. The most important of these drawbacks is that neither the programmer nor the compiler have control any more of the order in which alternatives are tried. This can be a problem in practice since in that case it is difficult to take advantage of the efficiencies of the "inner loops" method used for generating alternatives in backtracking systems: in practical systems the user (or the compiler) can minimize backtracking by correctly ordering the literals as a function of parameters such as the number of potential alternatives for each procedure call (the number of clauses in the procedure), the number of arguments and free variables in each call, and user knowledge about the problem's search space characteristics.

Another drawback related with a run-time dependent backtracking scheme from a practical point of view is the difficulty in obtaining performance figures for the model: since the amount of computation in order to obtain a solution depends on the order in which the search space is explored, in such a non-deterministic environment different execution times will be obtained for each run of the problem. Also, the amount of computation is different from that of a sequential system so that speedup figures are hard to obtain unless averages from many runs are computed. In this section an extension to the local goals execution model of the previous section is presented which avoids such problems at the expense of a small overhead. In particular, this model will be capable of preserving the same order as a sequential implementation in the generation of alternatives.

The reason for the change in the backtracking order in the **LGF** method is that there is no way of differentiating a normal *choice point* from those generated by "local goals". This can be seen in figure 6-4-C: suppose that **d** still has alternatives. If **e** now runs out of alternatives, the *choice point* corresponding to **b** will be tried next (instead of sending a "redo" message to **d**/*p5*) because there is no way to detect that this *choice point* corresponds to a "local goal" and that the *Parcall Frame* should be checked before trying the next alternative of **b** (if the conventional right to left backtracking order is to be preserved).

The above mentioned problem can be solved through the use of a series of "markers", which are stored in the *Stack* in very much the same way as conventional *choice points*. In fact, in this model (the "marker model" for **RGF** backtracking) *choice points* are just one more type of "marker" in the *Stack*. In **RGF** backtracking two types of markers (in addition to *choice points*) are used:

- **Wait Markers**: in the **RGF** model the *Parcall Frame* described for the **LGF** model is split into two parts, the *Parcall Frame* itself (containing the "slots", inside/outside flag, ...) and the *wait marker* (containing the pointers into the data areas and a pointer into the *Parcall Frame*). The *Parcall Frame* is still pushed on to the *Stack* as soon as the parallel call is entered, but the *wait marker* is only pushed on to the *Stack* upon exit from the parallel call (i.e. when execution of all goals within the call is completed). Thus, during backward execution, this marker will be found on the *Stack* above all local goals, and it will point to the *Parcall Frame* which can now be analyzed before any local goals are backtracked.

- **Local Goal Markers**: a *local goal marker* is pushed on to the *Stack* every time a "local goal" is picked up. It contains the values of the pointers into the data areas, a pointer to the *Parcall Frame*, and the slot id. for this goal. These markers are essentially equivalent to the "input goal markers" which were introduced in Chapter 5 for separating stack sections corresponding to the execution of different "remote goals", but applied to the execution of "local goals". They ensure that if a local goal fails, the *Parcall Frame* will be consulted (for example, to detect "inside" backtracking) before any other local goals in the *Stack* are backtracked.

The execution in an **RGF** system of the same example used in previous sections is illustrated in figure 6-5. In part A of this figure the contents of the *Stack* and *Heap* are shown after having executed *a1:* and entered the parallel call (pushing a *Parcall Frame* on to the *Stack*). **d** has been picked up by *P5* (and returned with no alternatives) and **b** has been executed locally. The local processor has just picked up **c** (and pushed the corresponding *local goal marker* on to the *Stack*) and is currently executing this goal. The *Parcall Frame* is still marked as "inside".

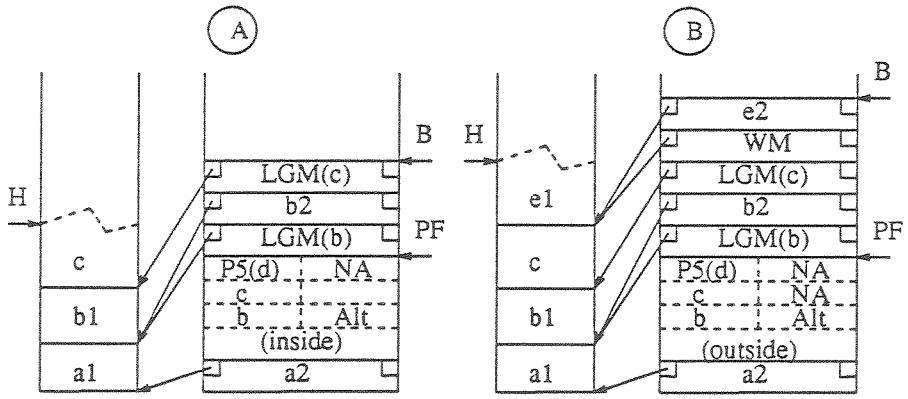

Figure 6-5: "Right Goals First" (**RGF**) Backtracking

In figure 6-5-B execution has proceeded after the success of **c** by pushing the *wait marker* on to the *Stack* and continuing with the first alternative of **e**. Note how now, if **e** runs out of alternatives, the first element on the *Stack* will be the *wait marker* (rather than the *choice point* for **b**) which will refer the backward execution algorithm to the *Parcall Frame*. Now, while scanning the *Parcall Frame*, a "redo" message could be sent to *p5* in case **d** had alternatives before backtracking **b**, thus preserving the right to left backtracking order.

If it is supposed that all "markers" (including *choice points*) are linked

together so that the last *marker* is always pointed to by **B**, the general algorithm for **RGF** backtracking is then[52]:

- If *Local Failure*, then:
  - If **B** > **PF** then
    - If **B** points to a *choice point*, perform the normal *choice point* backtracking.
    - If **B** points to a *Wait Marker* or a *Local Goal Marker*, perform *Parcall Frame Backtracking* on the *Parcall Frame* pointed to by the marker (this is the "failing *Parcall Frame*").
    - If **B** points to an *Input Goal Marker* (no *choice points* or *Parcall Frames* are left in this stack section), report failure to the parent.
  - If **PF** > **B**, perform *Parcall Frame Backtracking* on the *Parcall Frame* pointed to by **PF** ("failing *Parcall Frame*").
- If *Remote Failure*, then:
  - perform *Parcall Frame Backtracking* on the *Parcall Frame* referred to by the remote failure message ("failing *Parcall Frame*").

In the above described algorithm *"Parcall Frame Backtracking"* refers to the following series of actions:

- If the "failing *Parcall Frame*" is the same as the current one (i.e. the one pointed to by **PF**), then:
  - If the status of the *Parcall Frame* is "inside", "kill" all goals in the *Parcall Frame* and fail by recursively executing this algorithm in a *Local Failure* mode.

---

[52]Note that an additional entry is necessary then in each marker which contains its *type*. An alternative solution is to have different registers pointing at the last of each type of marker. The topmost marker and its type can then be identified by finding the register with the maximum value. This will be illustrated in the Parallel Abstract Machine design of the next chapter.

- If the status of the *Parcall Frame* is "outside", then:
  - Find the first slot (now scanning the slots in order, equivalent to right-to-left order in the clause) with pending alternatives whose corresponding process responds successfully to a "redo" message (when slots correspond to local goals with alternatives they are also tried ("redone"), but locally; previous slots are sent "unwind" messages).
  - If and when such a process is found, invoke the parallel execution of all the goals that correspond to the following slots.
  - If none succeeds, fail by recursively executing this algorithm in a *Local Failure* mode.

- If the "failing *Parcall Frame*" is different than the current one, then (note that this always corresponds to "inside" backtracking if the "point method" is used):
  - follow the chain of *Parcall Frames* up to and including the "failing" one (sending "unwind" messages to all remote processors in those slots),
  - fail by recursively executing this algorithm in *Local Failure* mode.

Note again that, although the description is lengthy because of the different cases involved, the abstract machine can select the appropriate case using arithmetic checks ($\mathbf{B} > \mathbf{PF}$ or $\mathbf{B} < \mathbf{PF}$; Status= 1 or 0; marker type = Local or Input or Wait) and the actions are in any case simple and determinate. Backward execution also proceeds in parallel (killing, unwinding of trails, etc.).

6.4 Chapter Summary

In the previous sections an efficient implementation scheme for distributed backtracking in Goal Independence models of AND-Parallelism has been presented which is a materialization of the algorithms offered in Chapter 4 in the framework of the memory management and goal scheduling model of Chapter 5. The concept of the *Parcall Frame*, methods for local execution of parallel goals, and some examples to illustrate their operation have been introduced. It is argued that this solution cleanly integrates distributed backtracking in one form of AND-Parallelism with the implementation technologies of high performance Prolog systems. A form of restricted intelligent backtracking is provided with virtually no additional overhead. "Soft" degradation of performance with resource exhaustion is attained: even a single processor will run any parallel program while still supporting restricted intelligent backtracking when goals are independent. In the next chapter these techniques will be materialized in the design of an Abstract Machine capable of AND-Parallel execution of Logic Programs. Regarding the choice of an execution methodology for "local goals", despite the additional overhead, the advantages of the "marker model" and **RGF** backtracking will make them the preferred choices in this design.

Chapter 7

An Abstract Machine for Restricted AND-Parallelism

This chapter describes an Abstract Machine and Instruction Set for parallel execution of Prolog programs annotated with Conditional Graph Expressions. Support is provided for both forward and backward execution of Goal Independent (Restricted) AND-Parallel calls as described in the previous chapters. The Abstract Machine basically represents an extension of the **WAM** to a parallel environment. In the next sections new data areas and abstract instructions will be defined which will be a materialization of the techniques and algorithms presented in the previous chapters. In much the same way as the WAM, *the design is "Abstract" in that certain details of the encoding and implementation are left open so that a practical realization can be made in a number of different forms.*

7.1 Extending the WAM for Parallel Execution

Most of the issues associated with the implementation of Goal Independence AND-Parallelism have been already dealt with in the previous chapters. The problem of extending the **WAM** to support this type of parallelism then basically entails providing additional mechanisms at the Abstract Machine level which will implement the various algorithms introduced there. Of course, this has to be done in an as efficient and unobtrusive as possible way, so that all the performance advantages of the **WAM** are retained. However, in the previous chapters several different alternative solutions were proposed for many of these issues. Some choices regarding these alternatives are outlined below.

Regarding the processor state diagrams introduced in Chapter 5, for simplicity, and unless otherwise noted, it will be assumed that there is only one (dual) process per processor comprising both the "foreground" and the "background" processes as described in that chapter. The interrupt mechanism sketched there, for example, can be used to implement this in a practical system[53]. A "Goal Restriction" model will be assumed regarding the scheduling strategy being used, so that a single set of stacks needs to be maintained per processor. This will assure simple and efficient memory management (all space being recovered during backtracking and always from the top of the stacks involved) as shown in Chapter 5. Regarding the execution of local goals in the implementation of distributed backtracking, the "marker" model of **RGF** backtracking, as described in Chapter 6, will be supported in the Abstract Machine[54]. Also, "point backtracking" (rather than "streak backtracking") as introduced in Chapter 4 will be assumed.

In view of the above mentioned assumptions, the particular issues which remain to be addressed in order to extend the sequential **WAM** for AND-Parallel execution can now be stated more concretely. Support has to be provided for the forward execution semantics described in Chapter 4: goal independence has to be detected (the conditions of the **CGE** checked), and, upon arrival at a parallel call (i.e. a **CGE** whose conditions evaluate to true), a scheduling mechanism such as that described in Chapter 5 has to assign available work (i.e. the parallel goals) to the

---

[53]In addition, conventional multiprocessing techniques can be used in order to support more than one of these (dual) processes in each processor: process swapping would then be used in place of wait states. Nevertheless, the "one process per processor" assumption will be used throughout the rest of the description since it is easier to explain and understand the model in these terms.

[54]An Abstract Machine which supports **LGF** backtracking has also been designed and simulated. This design is discussed in [31].

available processors. Thus, instructions have to be provided for pushing goals on to a *Goal Stack*, and the representation of these "goals" defined. Also, some data structure has to be provided to keep track of the state of execution of parallel siblings. The *Parcall Frame/Wait Marker* combination, as introduced in the previous chapter, will be used for this purpose. *Input Goal Markers* will be used to separate stack sections corresponding to different remote goals (i.e. goals received from another processor). *Local Goal Markers* will be used to mark the beginning of the execution of a local goal. Support also has to be provided for the (**RGF**) backward execution algorithm. This will be done with the aid of the *Parcall Frame* and the different markers as described in the previous chapter. The necessary "kill" and "unwind" messages will be handled by means of a small *Message Buffer*.

Figure 7-1 shows the data areas and registers for **one processing element** of the Parallel Abstract Machine. Each "processor" is essentially equivalent to a standard **WAM** except for the addition of a "*Goal Stack*" and the inclusion of "*Parcall Frames*" and "markers" in the *Stack*, together with *environments* and *choice points*. Also, the above mentioned *Message Buffer* is present. New registers are also provided to point to these new data structures. The details of these additions will be the subject of the next sections.

7.1.1 The Goal Stack

When the scheduling strategy was introduced in previous chapters, it was mentioned how each processor had a private *Goal Stack* where goals which were ready to be executed in parallel could be pushed on to. As seen in figure 7-1, each processor has a private *Goal Stack*. Each entry in the *Goal Stack* is called a *Goal Frame*. A *Goal Frame* contains all necessary information for remote execution of a goal. In particular, each *Goal Frame* contains the following items:

- **Procedure_name**: points to the first instruction of the procedure to be executed.

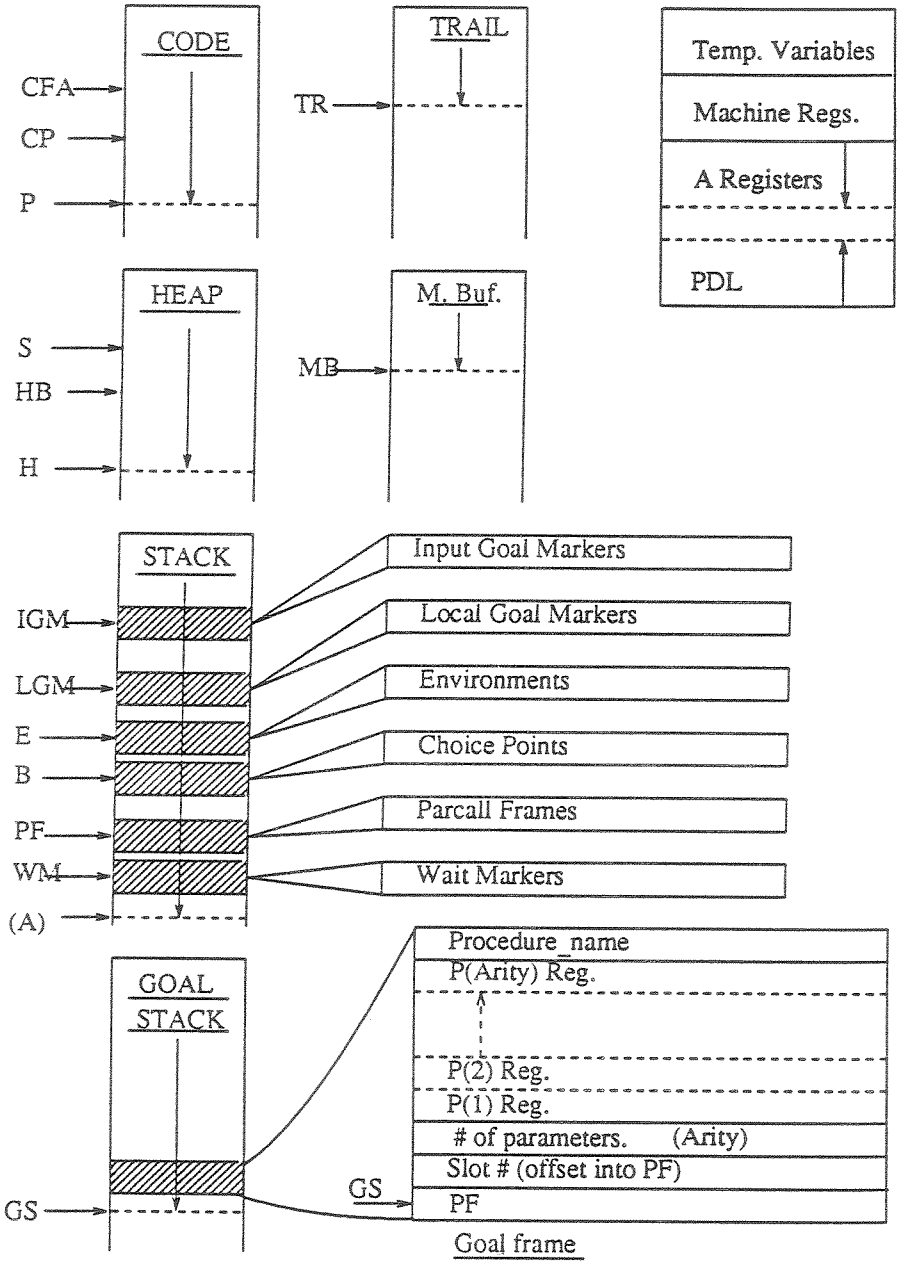

Figure 7-1: Data areas and registers: 1 processor, Parallel Abstract Machine

- **P(1),...,P(n) registers**: Parameter Registers. They are a copy of the n argument registers for the procedure.

- **#of parameters**: this cell contains "n", the *Arity* of the procedure.

- *Parcall Frame* **Pointer (PF)**: identifies which *Parcall Frame* this goal corresponds to.

- **Slot #**: identifies which slot in the *Parcall Frame* this goal corresponds to.

An extra machine register (**GS**) is also introduced. **GS** always points to the top of the *Goal Stack*. When a *parallel call* (a **CGE** whose "checks" succeed) is arrived at, all goals can be pushed on to the *Goal Stack*. In a shared memory environment these goals can then be directly "stolen" by "remote" processors from this "local" *Goal Stack*, provided a suitable memory arbitration technique is used (i.e. at least part of the *Goal Stack* has to be "locked" during this process). The "remote" processor, will then simply copy the parameter registers into its argument registers, load **P** with the address of "Procedure_name", and start execution from there.

A goal can also be picked up from its own *Goal Stack* by the **local** processor (the one which just pushed it there), using the same technique (while executing a `"pop_pending_goal"` instruction, to be described in the next sections). In this case the n parameter registers in the *Goal Frame* are simply copied into the local argument registers and execution continues as usual. The description of the `"pop_pending_goal"` instruction gives a complete account of the simple actions involved.

There is one more possible use for the **GS** register which was suggested in Chapter 5, in the implementation of the scheduling strategy: **GS** can be the value that is continuously being fed to the scheduling network. Its value effectively gives an estimation of the amount of parallel work available in the processor. If this scheme is used, then an idle processor will always receive from the sorting network the Pid. of the processor with the highest **GS** value. Goals will then always be picked up from the *Goal Stack* that has more entries at any given point in time (load balancing).

7.1.2 Parcall Frames

Entries in the *Goal Stack* completely disappear after they are "picked up" by remote processors. As mentioned in the previous chapter, an additional data structure is thus needed in the local processor in order to:

1. keep track during forward execution of the parallel activities of the children processors which "picked up" the goals inside a parallel call,

2. select the appropriate actions during backtracking.

The "*Parcall Frame*" introduced in the previous chapter will be used for these purposes. One *Parcall Frame* is created for each parallel call. For each goal available for execution in parallel (i.e. for each goal pushed on to the *Goal Stack*) within this parallel call, there is one *slot* in the *Parcall Frame*. Each one of these slots has the following fields:

- **Process Id.**: this field contains the id. of the processor which picked up the corresponding goal. If it was the local processor, this field is marked accordingly ("*").

- **Completion Status**: this is a one bit field, set by the corresponding processor when it returns, marking whether it still has alternatives or not.

- **Ready/NotReady**: this is also a one bit field, used (by the "`check_ready`" instruction) to select the goals that are actually going to be pushed on to the *Goal Stack*. It is used when only *some* of the goals inside a parallel call need to be scheduled, as is the case during forward execution after backtracking. When a *Parcall Frame* is created, all Ready bits in all slots are initialized to ready.

In addition to a variable number of "slots", some fixed entries are needed in the *Parcall Frame*:

- **# of goals still to schedule**: this cell is initialized to the number of goals to be executed in parallel. Each time the local or remote processors take a goal from the *Goal Stack* this number is decremented.

- **# of goals to wait on**: this cell is incremented by a remote processor when it "steals" a goal from the local *Goal Stack*. It is decremented every time a processor returns with success.

- **Total # of slots in the** *Parcall Frame*: determines the size of the *Parcall Frame*.

- **Put instructions pointer (PIP)**: this cell contains the address of the first instruction of the first goal in the parallel call and is used to start pushing goals again on to the *Goal Stack* after backtracking. This time though, only those goals whose Ready field is set will be pushed, since all others are skipped by the "`check_ready`" instruction in front of them. The backtracking algorithm determines which Ready bits are to be set (i.e. which goals will be restarted) and reinitializes the values of the "# of goals still to schedule" cell above to the appropriate value.

- **Status**: this cell marks whether execution of the parallel call corresponding to this *Parcall Frame* has already been completed once ("**outside**" status) or the first pass is still going on ("**inside**" status). This is used to select the type of backtracking.

- **GS'**: the top of the *Goal Stack* upon entry to the parallel call is saved in this cell so that it can be restored during (inside) backtracking.

- **CPF**: continuation **PF**. The value of **PF** before this *Parcall Frame* is created is saved here. It is used to reset **PF** after exiting the parallel call.

Parcall Frames are just one more type of object which resides in the local *Stack*, together with *environments* and *choice points*. **PF** is an extra machine register which always points to the current *Parcall Frame*.

7.1.3 Wait Markers

A *Wait Marker* is pushed on to the stack upon successful exit from a parallel call. A dedicated register (**WM**) always points to the last *Wait Marker* in the *Stack*. The solution suggested in the previous Chapter of pointing at all markers with register **B** can, of course, also be used. However, having different pointers for each type of marker simplifies certain operations such as responding to a "kill" message and

detecting "success with no alternatives". Since the backward execution algorithm for the solution using the **B** pointer was already explained in the last chapter, the multiple register approach will be illustrated as an alternative in this design. In a practical implementation, the choice between one or the other scheme will, of course, ultimately be determined by the number of physical registers available for a real implementation. Entries contained in a *Wait Marker* are:

- **WM'**: The previous value of the **WM** pointer. This is used to reset the **WM** register to point to the previous *Wait Marker* during backtracking.
- **BPF**: The value of the **PF** register. The appropriate *Parcall Frame* is recovered during backtracking by resetting this register.
- **Pointers Into the Data Areas**: Other registers which point into the data areas are also saved to be reset during backtracking: **H'**, **TR'**, **BCP**, and **BCE**.

7.1.4 Input Goal Markers

An *Input Marker* is pushed on to the stack when a processor "steals" a goal form another processor's *Goal Stack*. Therefore, *Input Markers* mark the separation between different stack sections corresponding to the execution of different goals "stolen" from other processors. *Input Markers* are pointed to by the **IGM** register and chained together. Entries contained in an *Input Marker* are:

- **IGM'**: The previous value of the **IGM** register. This is used to reset the **IGM** register to point to the previous *Input Marker* upon complete failure of a given goal (or after responding to a "kill" or "unwind" message).
- **PF/Slot**: The value of the **PF** register and the slot # in the *Parcall Frame* in the parent which this goal corresponds to. This is received in the *Goal Frame* and is used to report success or failure to the parent (by updating the parent's *Parcall Frame* or sending a "goal failure" message).
- **Pointers Into the Data Areas**: Other registers which point into the data areas are also saved to be reset during backtracking: **H'**, **TR'**, **BCP**, and **BCE**.

7.1.5 Local Goal Markers

A *Local Goal Marker* is pushed on to the stack when a processor picks up a goal form its own *Goal Stack* (through a "`pop_pending_goal`" instruction). They are similar to *Input Markers* but applied to local goals. *Local Goal Markers* are pointed to by the **LGM** register and chained together. Entries contained in an *Local Goal Marker* are:

- **LGM'**: The previous value of the **LGM** register. This is used to reset the **LGM** register to point to the previous *Local Goal Marker* upon complete failure of a given local goal (or after doing a local "kill" of goals as a result of "inside" backtracking).

- **PF/Slot**: The value of the **PF** register of the local *Parcall Frame* and the slot in this frame which this goal corresponds to. This is taken from the *Goal Frame* and is used to report success or failure (by updating the *Parcall Frame* or starting the "local failure" routine).

- **Pointers Into the Data Areas**: Other registers which point into the data areas are also saved to be reset during backtracking: **H'**, **TR'**, **BCP**, and **BCE**.

7.1.6 The Message Buffer

In the Parallel Abstract Machine most interaction between the different processing elements is done implicitly through the *Parcall Frames* and the *Goal Stack*/Scheduling Network (reporting of success, synchronization, scheduling of goals, etc.). However, there are certain actions which require an immediate response from a given processor and which therefore need an independent communication channel. Such is the case for example, when the execution in a given processor needs to be interrupted and discarded as a result of intelligent backtracking ("kill" messages). A message buffer is provided in each processor for this purpose. Any other processor can write a message into this buffer. The top of the message buffer is pointed to by register **MB**. As soon as $\mathbf{MB} > \mathbf{0}$, the processor is interrupted and the message or messages pending in the message buffer attended. The types of messages used are

listed below (the actions associated with the receipt of these messages are described in the next sections).

- **"kill"**: This message is received from the parent processor. Execution of the *current* goal is to be interrupted and all computations associated with this goal need to be discarded.

- **"unwind"**: Also received from the parent. All computations associated with the *last* goal need to be discarded (the processor is not currently executing this goal).

- **"failure"**: This message is sent from a child process to the parent indicating that no solution could be found for the goal received. The **PF** pointer for the *Parcall Frame* associated with this goal and its *Slot number* within the frame are included with the message.

- **"redo"**: This message is received from the parent when the last goal executed still has alternatives and a new alternative is needed.

7.2 General Operation of the Parallel Abstract Machine

As stated before, each "processor" (figure 7-1) (or each "process" in a multiprocessing environment) is equivalent to a standard **WAM** with a complete set of registers and stacks, but including the new "*Goal Stack*", the *Message Buffer*, and the addition of "*Parcall Frames*" and "markers" to *environments* and *choice points* in the local *Stack*. In addition to the new registers pointed out in the previous sections, there is an additional register into the *Code* area (**CFA** --"Check fail address") which points to the code which should be executed if the conditions in the **CGE** fail, i.e. the code corresponding to sequential execution. Note how each of these sets of registers and stacks is particular to a processor, although all other processors have shared access to at least the *Stack* and *Heap*. Obviously the code area could be shared by all processing elements but it will be supposed that each one of them has its own copy of the code.

As soon as a processor "steals" a goal (a *Goal Frame*) from another processor's *Goal Stack*, it creates an *Input Goal Marker* on its local *Stack* (thus separating the data structures corresponding to the execution of this goal from those of the previous one) and starts working on the "stolen" goal by loading its argument registers from the parameter registers in the *Goal Frame* and fetching instructions starting at the location (procedure address) received. The local stacks will then grow (and shrink) as indicated by the semantics of the standard **WAM** instructions it is executing. It will be the "local" processor for this instruction stream and its data areas will be the "local *Stack*", "local *Heap*", and "local *Trail*", etc. Note though, that the *environments* in its local *Stack* and the data structures in its local *Heap* will contain *references to the data areas of ancestor processors*. The character of these references will vary depending on the memory organization used in the underlying architecture (i.e. from absolute addresses for uniform addressing space, shared memory architectures to, for example, Pid./remote-address pairs for non-shared memories). Variable precedence relationships within each processor are kept using conventional methods (i.e. address comparison). Some mechanism has to be provided for determination of these relationships across processors, depending on the scheduling and memory management strategy chosen. For example, if the "goal restriction" scheduling strategy is used (Chapter 5) the relative precedence of the stack sections that the variables being bound belong to determines the relative seniority of these variables.

Also note that, although there might be reading conflicts (two or more processors trying to read the same memory location), there can be no data related writing conflicts if the **CGE**'s have been generated correctly. The ill-effects of reading conflicts on performance are much easier to avoid than those of writing conflicts, for example by using multiported memories and/or data caching. Also all

synchronization is guaranteed by the wait instructions marking parallel call boundaries. This will become more clear after the instruction set has been introduced and an example commented on, but it shows how all program or data dependent control and synchronization issues are concealed within the semantics of the **CGE**'s.

When a parallel call is reached, a *Parcall Frame* is created in the local *Stack* and its goals are pushed on to the *Goal Stack*, ready to be picked up by the local processor or other remote processors. These remote processors will in turn work on their assigned goals operating on their own stacks and again possibly including references to ancestor stacks. Parallel goals can also be executed locally, creating the corresponding *Local Goal Marker*. As soon as all goals in the parallel call succeed, a *Wait Marker* is pushed on the *Stack* and execution can continue normally beyond the parallel call. Eventually, if all execution related to the "stolen" goal terminates successfully, this success will be reported to the parent by updating the corresponding slot in the *Parcall Frame*. Of course, there may be some entries (for example *choice points*, if the goal still has alternatives) left in the local *Stack*, some data structures in the local *Heap* that ancestors may need to access (the "output" of the procedure), and also some entries in the *Trail*. This is left this way (rather than doing any copying to the parents' data areas), and when the next goal is received its data structures can be grown above these. This space can still be retrieved if a kill message is received from the parent processor (because of a failure there or in some other related processor), much in the same way as in the sequential **WAM**. If, on the other hand, execution of the "stolen" goal ends in failure, this fact is reported to the parent and all storage used by this processor and all dependent processors is deallocated (up to and including the *Input Goal Marker*). Complete retrieval of storage on backtracking is thus achieved, also much in the same way as in a sequential implementation.

Note that, as pointed out in Chapter 5, if the relative precedence of stack sections is preserved, then "kill" and "unwind" messages necessarily always refer to the last goal executed (i.e. to the last set of structures on the *Stack* and *Heap*) and space is always retrieved from the top of the *Stack* or *Heap* as in the sequential model. Of course local unwinding of the *Trail* will now also undo any bindings done outside the local data areas. Both the recovery of storage and this unwinding of the distributed *Trail* is done completely in parallel by all the AND-siblings which receive "kill" messages. This is a source of parallelism during backward execution for this model. Also, note that with the above mentioned ordering of events, a "redo" message, when received, also always refers to the last *choice point* (or *Parcall Frame/Wait Marker* in the local *Stack* and it can be executed just as if a local failure had occurred!

One last observation: as mentioned in previous chapters, there is in general no point in pushing **all** goals in the parallel call on to the *Goal Stack*. If all goals are picked up immediately the local processor is left in wait mode with nothing to do, or a relatively high overhead process swapping immediately occurs if multiprocessing is implemented. If some goal remains in the *Goal Stack* it will be picked up by the local process but unnecessary work will be done in copying the argument registers and all other information to the goal stack and back to the local registers immediately after. A much better alternative is to always leave one of the goals in the parallel call (for example the last one) for local execution.

7.3 The Extended Abstract Machine Instruction Set

This section will describe the instruction set for the Parallel Abstract Machine. All **WAM** instructions are supported in addition to the new instructions implementing AND-Parallelism. The **WAM** instructions are listed first, then the new instructions with their related actions in the abstract machine, and finally the semantics of certain situations, such as failure, which are not directly coded as instructions are defined. Note how, although "`check_...`" instructions are somewhat particular to the implementation of **RAP**, all other instructions are appropriate for any Goal Independence AND-Parallel system.

The description of the instructions will be somewhat brief in order to facilitate later reference. Fully commented examples of their use can be found at the end of the chapter and in the appendices. Again note that the design is abstract in that some details have been left to be determined at implementation time. Thus, alternate implementations may extend or change the precise meaning of some of the instructions from that offered here.

7.3.1 WAM Instructions

As stated before, all **WAM** instructions are supported. Their operation will not be described in this section, because, except for the "`proceed`" instruction, it is the same as in their conventional interpretation as described by Warren [88]. The new semantics of the `proceed` instruction will be given in the next sections. Here is a list of the basic **WAM** instruction set for reference (built-ins and special instructions not directly listed in Warren's report such as those dealing with arithmetic, "cut", cdr-coding etc. are omitted):

Summary of WAM Instructions

	HEAD	BODY
procedural		
	(proceed)	execute P
		call P,N
	allocate	deallocate
get/put		
	get_variable Xn,Ai	put_variable Xn,Ai
	get_variable Yn,Ai	put_variable Yn,Ai
	get_value Xn,Ai	put_value Xn,Ai
	get_value Yn,Ai	put_value Yn,Ai
		put_unsafe_value Yn,Ai
	get_constant C,Ai	put_constant C,Ai
	get_nil Ai	put_nil Ai
	get_structure F,Ai	put_structure F,Ai
	get_list Ai	put_list Ai
unify		
	unify_void N	
	unify_variable Xn	
	unify_variable Yn	
	unify_local_value Xn	
	unify_local_value Yn	
	unify_value Xn	
	unify_value Yn	
	unify_constant C	
	unify_nil	
indexing		
	try_me_else L	try L
	retry_me_else L	try L
	trust_me_else fail	trust L
	switch_on_term Lv, Lc, Ll, Ls	
	switch_on_constant N,table	
	switch_on_structure N,table	

7.3.2 Check Instructions

"Check" instructions are used to encode the "conditions" in a **CGE**. Two types of checks ("ground" and "independent") and a branch instruction are provided. Note that by combining these, any kind of disjunctions or conjunctions of checks on any number of variables can be expressed (this is shown in the appendices):

check_me_else Label

- load check failure address with Label (**CFA**=Label).

check_ground Vn

- dereference register Vn and check to see if its contents are ground. If so, continue with next instruction; otherwise **P=CFA** (i.e. branch to Check Failure Address).

check_independent Vn,Vm

- dereference Vn and Vm. If they are independent, next instruction; otherwise **P=CFA**.

As stated when **CGE**s were introduced in Chapter 5, the particular algorithm used to check for independence is left as an implementation issue. Again, one algorithm which can be used is DeGroot's [25] where two variables are independent if at least one of them is ground or if they are both uninstantiated variables which do not dereference to the same location (i.e. they do not "share"). Other algorithms can also be used as long as they evaluate independence in a conservative way: it must be ensured that the algorithm never renders two dependent variables as dependent (since this would lead to variable binding conflicts at run-time) although, on the other hand, it may well prove advantageous to implement a fast algorithm (such as DeGroot's) which sometimes gives up on checking complex terms or long dereferencing chains (by considering them immediately as dependent), even though it may thus miss some opportunity for parallelism.

A possible optimization to this scheme which can avoid the checking

overhead when the system is loaded is to check whether the *Goal Stack* is already full or above a certain threshold and in that case prevent the creation of parallel processes regardless of whether the conditions evaluate to true or not (i.e. if goals are not being picked up by other processes because everybody is busy there is not much point in generating work). This can be easily implemented by changing the semantics of the "check_me_else" instruction to the following:

- if **GS** is above threshold, jump to Label;
- otherwise load check failure address (**CFA**) with Label.

Now execution always jumps to Label if the value of **GS** is above a certain threshold. Note, though, that in this case semi-intelligent backtracking will also be prevented from working in all these cases, while if goals in a parallel call are always pushed on to the *Goal Stack* then semi-intelligent backtracking is still supported even if they are all executed locally. Therefore, implementation of this feature is only recommended for the case when the *Goal Stack* is actually full.

7.3.3 Goal Scheduling Instructions

These are the instructions used for pushing goals with their arguments on to the *Goal Stack* and for picking up these goals in the local processor:

push_call Procedure_name/Arity,Slot#

- request exclusive access to *Goal Stack*;
- push on to the *Goal Stack*: "Procedure_name", registers A_{Arity}, $A_{Arity-1}$, ...A_1, "Arity" ("n"), Slot# (i.e. offset from **PF** for the slot corresponding to this goal), and current **PF** pointer;
- release access to *Goal Stack*.

The arguments should be first loaded into the argument (A) registers using normal "put_..." instructions (as for a conventional "call"). Then, they will be

transferred in one section to the *Goal Stack* with the `push_call` instruction. Of course the arguments could be directly "put" in the *Goal Stack* by special "`put_...,Pn`" instructions, but note that then the *Goal Stack* has to be locked until the `put/push_call` sequence is completed (since there is always an incomplete goal on top of the stack). This leaves little opportunity for any picking of goals before all goals are pushed on to the stack. In the approach chosen the goal stack can be accessed freely by other processors during the "put" sequence, and it is only locked during the `push_call`. It also has the advantage of avoiding a new set of put instructions.

pop_pending_goal

- if no goals are pending to be scheduled ("# of goals to schedule" in *Parcall Frame* = 0), continue with next instruction;
- else pop a goal from the local *Goal Stack* (described below).

This instruction is used by the local processor to pop a goal from its own *Goal Stack* for local execution. A *Local Goal Marker* is created on the local *Stack* (and **LGM** updated), the corresponding slot in the *Parcall Frame* (as indicated by "Slot #" in the *Goal Frame*) is marked as "local", the "# of goals still to schedule" is decremented, and the arguments are popped back from the *Goal Stack* into the local argument registers. Then **P** is loaded with the address of "Procedure_name" and execution continues from there. The continuation pointer (**CP**) is *set to point to the Input Goal Marker*. When this goal finally succeeds, the "`proceed`" instruction will detect that success of a local goal has just occurred (the **CP** pointer being out of the program area) and will update the slot in the *Parcall Frame* (a pointer to it is stored in the *Local Goal Marker*). Execution will then continue at **PIP** (i.e. the beginning of the "`put, push_call`" sequence, stored in the *Parcall Frame*). This sequence, except in the case of success after "redoing" the local goal (when other goals will then need to be pushed on to the *Goal Stack* for continuing forward execution),

will be skipped and the "pop_pending_goal" executed again. Thus, any other pending goals will also be popped from the *Goal Stack*. This process continues until there are no more goals left (# of goals to schedule = 0). The next instruction is then executed.

An optimization can be implemented which can help avoid the additional overhead involved in creating "markers" (this optimization is also applicable to *Wait* and *Input Markers*). Since most of the information contained in a "marker" is also contained in a *choice point*, if a *choice point* is created immediately after a marker they can be combined into an "extended choice point". This is the case, for example, if a local goal is popped and the corresponding *Local Goal Marker* created, and the first instruction which accesses the data areas in the execution of the goal is a "try_me_else" instruction. The marker can be then extended (by including the argument registers, **B'**, and **BP**) and serve both purposes.

7.3.4 Control Instructions

These instructions take care of the control issues involved in a parallel call: creating and deleting *Parcall Frames* and *Wait Markers*, selecting the goals to schedule, and waiting for children to report results.

allocate_pcall #_of_slots,M

This instruction creates a properly initialized *Parcall Frame* in the local *Stack* with the correct number of slots. M, the number of "permanent variables" still needed in the *environment*, is used to extend the concept of *environment trimming*. **PF** now points to the top of the stack. The actions involved in creating this Frame are:

- **CPF = PF** (save continuation *Parcall Frame* pointer)
- push on to the stack in order:

  - "**#_of_slots**" initialized cells (Pid=nil, CompStatus=nil, Ready/NotR=Ready) one for each goal in the *Parcall Frame*.

  - "**#_of_slots**" (number of goals still pending to be scheduled for this frame).

  - "**#_of_slots**" (number of goals to wait on before exiting the pcall).

- push **P** (which points to the first instruction of the `check/put/push_call` sequence since it is always the next instruction to the `allocate_pcall`). This value is called "**PIP**" ("put" instructions pointer).

- push Status, initialized to "inside".

- push **GS** (to be restored upon backtracking).

M is used to extend the concept of environment trimming: if the last object on the stack is an environment with N>M permanent variables, and only M are needed from now on, the *Parcall Frame* can be pushed in the stack immediately after the first M valuecells in the environment, thus discarding the N-M unneeded cells.

check_ready Slot_#,Label

- Check that slot in the current *Parcall Frame* (pointed to by **PF**).

- If the slot status is "not ready", jump to Label;

- else, set the slot status to "not ready" and continue with the next instruction.

`check_ready` instructions are used to skip those goals whose slots are marked as "NotReady" in the *Parcall Frame* so that they are not pushed on to the *Goal Stack*. This is useful during backtracking, as only some of the goals inside a parallel call may need to be restarted after failure.

wait_on_siblings

- wait until "# of goals to wait on" in current *Parcall Frame* is 0;
- then, push a *Wait Marker* on to the *Stack* (saving current **PF** in **BPF**),
- restore **PF** from the *Parcall Frame* (**PF=CPF**),
- change status to "**outside**" (if it is "**inside**"),
- go on to next instruction.

Note that this "wait" only implies an idle processor if no multiprocessing is actually implemented. If multiprocessing is available then a ready process can be paged in at this point, or if all processes are in wait mode then a new process can be created which will start as idle, asking other processors for work through the sorting network, as described in Chapter 5. Nevertheless, in an implementation where there is a fairly large number of simple processors it may prove advantageous to simply let the processor wait. This is due to the fact that there is a hierarchical relationship between processes. If a waiting processor starts a new process in it to avoid a wait state it will most likely pick up a goal from one of its descendants. If immediate action is needed then on the original waiting process (for example handling a failure report from a child) the extra time necessary to swap processes again in order to service it will add to the execution time of the "main line of processing" while the work done since the wait is useless because all children have to be killed anyway. If the processor had been simply waiting response could have been immediate. The average ratio of successes vs. failures in representative programs will determine the

extent of this effect[55].

An extension of last call optimization (and of the creation of choice points only when needed) can be implemented in the `wait_on_siblings` instruction by discarding the current *Parcall Frame* (**PF=CPF**) and *not creating a Wait Marker* if all slots in the frame are marked as having "no alternatives". In order to understand this, note that after the `wait_on_siblings` instruction the *Parcall Frame* and *Wait Marker* would only be needed in the event of having to backtrack any of the goals inside the parallel call. If none of the goals have any alternatives then none of them need to be backtracked, and, if the *Parcall Frame* and *Wait Marker*[56] have been discarded, then if failure occurs, execution would simply return to the first *Choice Point* (or *Parcall Frame/Markers before* this *Parcall Frame* which would be the correct backtracking point at this time.

Care should be taken though, when implementing this feature in some special cases. Note that even in the case when there are some processors with no alternatives, these processors *may still have portions of their trail which would have to be "unwound" during backtracking.* One way of solving this problem is by including a new field in each goal slot in the *Parcall Frame*, where processors with no alternatives

---

[55]In fact, it is also possible to completely avoid wait states without introducing multiprocessing by letting any processor in the *Parcall Frame* which happens to be the last one to complete execution of the last goal in the parallel call (instead of only the local (parent) process) to pick up the continuation (i.e. the execution of the rest of the body of the clause past that parallel call). Thus the parent processor can be free to look for a remote goal if there are no more local goals left to execute (rather that wait for completion of all siblings to pick up the continuation). However, the implications of this strategy in other areas of the design (such as the memory management) have not been studied and are therefore left as a subject of future research.

[56]... and the *Local Goal Markers* of all local goals with no alternatives.

mark if they have a pending segment of the trail to be unwound upon backtracking or not. Then the *Parcall Frame* is only discarded if all slots are marked both as "no-alternatives" and "no-trail".

A better solution yet is to go ahead and discard the *Parcall Frame* even if some remote processors have pending trail segments, but *push a special entry in the local trail with the Processor id. of each such processor.* Then as the local trail is unwound these entries are identified and "unwind" messages are sent to the corresponding Processors. This approach has the additional advantage that the "unwind" messages will also take care of discarding the space still being used in the heap so that total space recovery after backtracking is maintained in addition to last call optimization. Other solutions such as independent local copies of the remote trail can be considered as alternative implementation schemes, but they will probably result in undesirably high communication traffic.

7.3.5 Modified Instructions

The **proceed** instruction of the **WAM** instruction set needs to be modified in order to detect goal success and to perform the corresponding reporting to the parent:

proceed

- If **CP** "not special" (i.e. **CP** points into the *Program Area*), **P** = **CP**.
- If **CP** points to the *Input Goal Marker* (**CP** = **IGM**), then execution of a remote goal has succeeded:
  - get **PF** and "Slot #" from the *Input Goal Marker*,
  - update this slot with Success with or without alternatives,
  - decrement "# of goals to wait on",
  - return to idle loop.

- If **CP** points to a *Local Goal Marker*, then execution of a local goal has succeeded:
  - get **PF** and "Slot #" from the *Input Goal Marker*,
  - update this slot with Success with or without alternatives,
  - get **PIP** from the *Parcall Frame*,
  - **P** = **PIP** (execution continues at the `put/ push_call/ pop_pending_goal` sequence).

Note that the detection of whether a goal has alternatives or not can be simplified to a comparison of registers **B**, **PF**, and **IGM**: a goal has no alternatives when **PF** $<$ **IGM** and **B** $<$ **IGM**.

7.3.6 Other Non-Instruction Related Actions

In addition to the operations associated with particular instructions, each processor has to support other actions resulting from exceptions such as messages arriving from other processors or failure. These actions obviously differ somewhat from the corresponding ones in a sequential implementation. We will sketch some of them in this section.

failure

The actions required during failure for the "marker" model of **RGF** backtracking were already given in the previous chapter. The following algorithm is essentially equivalent to that of Chapter 6 but taking advantage of the multiple register approach taken in this design (**B**, **PF**, **IGM**, **LGM**, **WM**) and taking into account some of the optimizations suggested in this chapter. Again, a distinction is made between *Local* and *Remote Failure*:

- *Local Failure*: failure originates within the local processor (i.e. during a unification being performed in the local processor).
- *Remote Failure*: a "failure" message is received from a child process.

The algorithm, then, follows:

- If *Local Failure, find the maximum (*MAX*) of* **B**, **PF**, **IGM**, **LGM**, **WM** (i.e. find "marker" on top of the stack), then,
  - If MAX = **B**, then perform the normal *choice point* backtracking (reset registers, "unwind" *Trail*, continue with next alternative).
  - If MAX = **LGM** (i.e. a *Local Goal Marker* is the last marker on the *Stack*), reset pointers into data areas, "unwind" *Trail* and perform *Parcall Frame Backtracking* on the *Parcall Frame* pointed to by the marker (this is the "failing *Parcall Frame*").
  - If MAX = **WM** (i.e. a *Wait Marker* is the last marker on the *Stack*), reset pointers into data areas, "unwind" *Trail*, *reset* **PF** *from* **BPF** and perform *Parcall Frame Backtracking* on the *Parcall Frame* pointed to by the marker (this is the "failing *Parcall Frame*").
  - If MAX = **PF**, perform *Parcall Frame Backtracking* on the *Parcall Frame* pointed to by **PF** ("failing *Parcall Frame*").
  - If MAX = **IGM** (i.e. an *Input Goal Marker* is the last marker on the *Stack*; no *choice points* or *Parcall Frames* are left in this stack section)), goal failure: reset pointers into data areas, "unwind" *Trail*, send a "failure" message to the parent (including the **PF**/Slot # from the *Input Goal Marker*) which will execute the "remote failure" routine, return to `idle` loop.
- If *Remote Failure*, then:
  - perform *Parcall Frame Backtracking* on the *Parcall Frame* referred to by the remote failure message ("failing *Parcall Frame*").

In the above described algorithm *"Parcall Frame Backtracking"* refers to the following series of actions:

- If the "failing *Parcall Frame*" is the same as the current one (i.e. the one pointed to by **PF**), then:

- If the status of the *Parcall Frame* is "inside", send "kill" ("unwind" if execution has completed) messages to all processors corresponding to remote goals in the *Parcall Frame* and fail by recursively executing this algorithm in a *Local Failure* mode.

- If the status of the *Parcall Frame* is "outside", then:

  - Find the first slot (scanning the slots in order, equivalent to right-to-left order in the clause) with pending alternatives whose corresponding process responds successfully to a "redo" message (when slots correspond to local goals with alternatives they are also tried ("redone"), but locally).

  - If and when such a process is found, invoke the parallel execution of all the procedure goals that correspond to the following slots by setting the status in these slots to "ready" and branching to **PIP** (thus those goals will be pushed on to the *Goal Stack* and executed in parallel).

  - else, if none succeeds, fail by recursively executing this algorithm in a *Local Failure* mode.

- If the "failing *Parcall Frame*" is different than the current one, then (note that this always corresponds to "inside" backtracking if the "point method" is used):

  - follow the chain of *Parcall Frames* up to and including the "failing" one (by following the chain of *Wait Markers*) sending "kill"/"unwind" messages to all slots corresponding to remote processors,

  - fail by recursively executing this algorithm in *Local Failure* mode.

Note that, because of the deallocation of *Parcall Frames*, special entries can be found in the *Trail* which refer to processors with pending portions of the *Trail* and the corresponding "unwind" messages have to be sent to them. Also note that all "markers" are discarded after the pointers into the data areas are reset from them and the *Trail* unwound to the point saved in the "marker".

kill

"kill" is a message which can arrive from the parent processor indicating that the goal being solved in the local processor is not useful any more and should be discarded. The chain of *Parcall Frames/Wait Markers* is followed sending "unwind" messages (or "kill" messages if they are still running) to all children processors and "unwinding" the *Trail* until no *Parcall Frames* are left above the current *Input Goal Marker*, the pointers into the data areas are then restored from this marker, and the processor returns to idle (i.e. looking for work).

unwind

this message is sent by the parent when backtracking, and is equivalent to a "kill" message (except that the processor is not executing the referred goal at the time). Again all storage and children processes are discarded up to and including the current *Input Goal Marker*.

redo

redo is also received from the parent processor after reporting a solution which had a *choice point* available (i.e. after reporting "success with alternatives"). It is executed just as if local failure had occurred: go to the first *choice point*/**PF**/"marker" on the *Stack*, etc.

idle

This pseudo-instruction represents the idle loop in which a processor consults the scheduling network (or other processors' *Goal Stacks* if the scheduling network approach is not implemented) in order to find the id. of a processor which has available goals in its *Goal Stack*. As soon as such a processor is found a `pop_foreign_goal` pseudo instruction is executed.

pop_foreign_goal Pid./GS

This is equivalent to the `pop_pending_goal` instruction, but applied to a remote goal:

- pop a goal from Pid.'s *Goal Stack*.

This pseudo-instruction is used by a previously idle processor to pop a goal from a remote *Goal Stack* for execution. An *Input Goal Marker* is created on the local *Stack* (and **IGM** updated), the corresponding slot in the *Parcall Frame* (as indicated by "Slot #" in the *Goal Frame*) is updated with this processor's Pid., the "# of goals still to schedule" is decremented, the "# of goals to wait on" incremented, and the arguments are popped back from the *Goal Stack* into the local argument registers. Then **P** is loaded with the address of "Procedure_name" and execution continues from there. The continuation pointer (**CP**) is set to point to the *Input Goal Marker*. When this goal finally succeeds, the `"proceed"` instruction detects this "special" value in the continuation pointer, reports success to the parent by updating the *Parcall Frame*, and returns to the idle loop (looking for another goal to work on).

7.4 An Example

This example illustrates the code generated by the compiler for a simple clause[57]. The comments provided explain the operation of the instructions involved. Given the following "Prolog" clause[58]:

```
f(X,Y,Z):- a(X,Y), b(X,Y), c(X,Y), d(X,Y,Z), e(X,Y,Z).
```

the Graph Expression generated by the compiler after its analysis might be:

---

[57] Other examples can be found in Appendix 1.

[58] This clause is purposedly chosen so that the code generated is as simple as possible (no "unsafe variables", no special unification instructions) in attention to the reader with no previous exposure to **WAM** code. Also some of the instructions are obviously unnecessary but leaving them there makes it easier to visualize the structure of the code.

```
f(X,Y,Z):- a(X,Y), (ground(X,Y) | b(X,Y) & c(X,Y) & d(X,Y,Z)), e(X,Y,Z).
```

Clearly, in this clause it is expected that **a** will ground **X** and **Y**. In this case, "**ground(X,Y)**" will succeed and then **b**, **c**, and **d** will be able to run in parallel. Otherwise they will run sequentially and the annotated clause will execute the same instructions as the original one would have in a conventional system. The code that the compiler would generate for the clause above follows. In order to understand the first instructions in the example, recall that at the point of entering this code the calling procedure has already loaded registers A_1, A_2, and A_3 with the arguments for f:

```
     ------------------------------------------------------------------------
f/3:                          | (Entry point for procedure f)
     allocate                 | Push an environment on to the stack. It will
                              | have space for "X"(Y3), "Y"(Y2) and "Z"(Y1)
     ------------------------------------------------------------------------
                              | HEAD INSTRUCTIONS: f(X,Y,Z):- ...
     get_variable  "X",A1     | X <- (A1)  Unify (just "get" in this case) the
     get_variable  "Y",A2     | Y <- (A2)  arguments (X,Y,Z) from the parameter
     get_variable  "Z",A3     | Z <- (A3)  registers into the environment.
     ------------------------------------------------------------------------
                              | BODY INSTRUCTIONS: ... :- a(X,Y), ...
     put_value "X",A1         | (X) -> A1  Load argument registers from the
     put_value "Y",A2         | (Y) -> A2  environment for a.
     call  a/2,3              |            Call a.
     ------------------------------------------------------------------------
                              | ... ( ground(X,Y) | ...
     check_me_else SEQ_CODE   | Set the address to branch to in
                              | case the conditions fail (CFA).
     check_ground  "X"        | X ground?  if not go to SEQ_CODE
     check_ground  "Y"        | Y ground?  if not go to SEQ_CODE
     ------------------------------------------------------------------------
                              | The checks succeeded: parallel execution.
     allocate_pcall 3,3       | First, create a Parcall Frame in the stack with
                              | 3 slots (slot 1 for b, 2 for c, 3 for d)
                              | (3 is # of perm. vars. -used for env. trimming)
     ------------------------------------------------------------------------
P_I_P:                        | ... & d(X,Y,Z) ) ...
     check_ready 3,POP        | See if slot 3 in Parcall Frame (i.e. "d") is
                              | ready (always true except when backtracking);
                              | else, jump to POP (skip rest of goals)
     put_value "X",A1         | (X) -> A1   Load argument registers from the
     put_value "Y",A2         | (Y) -> A2   environment for d.
     put_value "Z",A3         | (Z) -> A3
     push_call d/3,3          | Push call to "d" with its arguments on to Goal
                              | Stack (it can now be "stolen" by another proc.)
                              | (Calls are pushed in reverse order if a Goal
     check_ready 2,POP        | STACK is used)
     put_value "X",A1         | ... & c(X,Y) & ...
     put_value "Y",A2         | (same as d above)
```

```
     push_call c/2,2          |
     ------------------------------------------------------------------------
                              |
     check_ready 1,POP        |
     put_value "X",A1         | ... | b(X,Y) & ...
     put_value "Y",A2         | (same as c above)
     push_call b/2,1          |
     ------------------------------------------------------------------------
POP:                          |
     pop_pending_goal         | If no goals pending, next instruction; else
                              | execute remaining goals locally (create LGM)
     wait_on_siblings         | Wait until all "remote" goals in the
                              | Parcall Frame have returned/create WM
     execute  CALL_E          | Go on to execute "e" (CALL_E).
     ------------------------------------------------------------------------
SEQ_CODE:                     | Checks failed: sequential execution.
     put_value  "X",A1        | (X) -> A1   Normal WAM code for executing b,
     put_value  "Y",A2        | (Y) -> A2   c, and d sequentially.
     call  b/2,3              | call "b".
                              |
     put_value  "X",A1        | (X) -> A1
     put_value  "Y",A2        | (Y) -> A2
     call  c/2,3              | call "c".
                              |
     put_value  "X",A1        | (X) -> A1
     put_value  "Y",A2        | (Y) -> A2
     put_value  "Z",A3        | (Z) -> A3
     call  d/3,3              | call "d".
     ------------------------------------------------------------------------
CALL_E:                       | "Normal" WAM call to "e".
     put_value  Y3,A1         | (X) -> A1
     put_value  Y2,A2         | (Y) -> A2
     put_value  Y1,A3         | (Z) -> A3
     deallocate               | Discard environment: last call optimization.
     execute  e/3             | Execute "e".
```

7.5 Determinate Execution

A set of alternative instructions is provided which supports the determinate execution algorithms introduced in Chapter 4. Some examples of the use of these instructions are given in the appendices. The operations involved in the execution of these instructions are similar to those of their previously introduced counterparts, but advantage is taken of the knowledge of the fact that only one solution is needed from the associated goals. Also, details of the operation of previous instructions are modified by the existence of determinate goals. Some of these new instructions and changes are sketched below.

7.5.1 Goal Scheduling Instructions

push_det_call Procedure_name/Arity,Slot#

- request exclusive access to *Goal Stack*;
- push on to the *Goal Stack*: "Procedure_name", registers A_{Arity}, $A_{Arity-1}$, ...A_1, "Arity" ("n"), Slot# (i.e. offset from **PF** for the slot corresponding to this goal), and current **PF** pointer;
- *mark the goal as "determinate"*;
- release access to *Goal Stack*.

The inclusion of the fact that the goal is determinate is very useful for the processor "stealing" the goal: it is known that this goal will never need to be redone. Among other issues, and as pointed out in Chapter 5, this knowledge widens the choice of goals (during the `idle/pop_foreign_goal` pseudo-instructions) which can be stacked above this goal (since in the worst case only the "garbage slot problem" can appear).

pop_pending_goal

If the goal picked up locally is determinate, *no Local Goal Marker needs to be created* and no updating of the *Parcall Frame* is necessary (except decrementing the "# of goals to schedule" field). The continuation can be set directly to this same instruction (i.e. to the current **P**), and no special action has to be taken in the proceed instruction.

7.5.2 Control Instructions

allocate_det_pcall #_of_slots,M

In a determinate *Parcall Frame* the slots only need to contain the Pids. of the remote processors which have executed goals within the call. Other than the slots, only the **CPF**, "# of goals to schedule", and "# of goals to wait on" fields are needed.

check_ready Slot_#,Label

`check_ready` instructions are not needed in determinate execution, since no backtracking takes place.

cut_merge

This instruction replaces the `wait_on_siblings` instruction for determinate execution:

- wait until "# of goals to wait on" in current *Parcall Frame* is 0;
- then, "trail" the Pid. contained in each slot corresponding to a goal executed remotely,
- restore **PF** from the *Parcall Frame* (**PF=CPF**) (i.e. discard the *Parcall Frame*)
- (discard all local *Choice Points* until before the *Parcall Frame*[59]).
- go on to next instruction.

Note that "last call optimization" is always implemented for determinate execution.

7.6 Performance Evaluation

An implementation of the abstract machine described in this chapter on a shared memory multiprocessor has been studied through simulations. These simulations have proved the functionality of the model and provided detailed information about its performance which have in turn affected several areas of the design. The main conclusion from the performance study in the efficiency of the Abstract Machine: if the problem being studied has intrinsic goal independence parallelism, the model will take advantage of it with very little overhead, typically

[59] If a real "cut" is needed, then all *choice points* and markers *up to the calling goal* should be discarded.

less than 10%. Speedup and "idle" times have been shown to be very favorable and determined again by the amount of intrinsic parallelism in the problem rather than by inefficiencies in the Abstract Machine design. In a system where processor utilization is of key importance it may prove advantageous to implement some way of dealing with "wait" times, such as multiprocessing or the "continuation" method suggested in this chapter. Details about the simulator and the simulations performed and definitions for the terms referred to above are given in Appendix B.

7.7 Chapter Summary

In the previous sections an AND-Parallel Abstract Machine design has been presented which is based on combining the techniques used in the **WAM** with the models of Goal Independence AND-Parallelism introduced in previous chapters. The same abstract machine and basic instruction set could also support with minor modifications other related AND-Parallel models and serve as a target for compilation of a variety of logic programming languages. The functionality and favorable performance of the design have been determined through simulations. It argued that this Abstract Machine is an attractive vehicle for the implementation of AND-Parallelism in the presence of "don't know" non-determinism: the compatibility with conventional **WAM** code makes sequential speed almost identical to that of the **WAM** and permits the use of current **WAM** compiler technology while also supporting the parallel execution algorithms described in the previous chapters. Simultaneously, most **WAM** optimizations are still supported, even during parallel execution, resulting in an efficient model. In particular, the scheduling and memory management techniques used ensure that the memory requirements of the parallel system are essentially equivalent to those of a sequential system (total storage) and that this space is still recovered during backtracking, thus effectively delaying the occurrence of garbage collection. A limited form of "intelligent backtracking" is

provided at very low overhead. The design also offers user-transparent distributed control and "soft" degradation of performance with resource exhaustion.

Chapter 8

Conclusion

In the previous chapters an efficient and reasonably complete parallel execution model for goal independence models of Parallelism for Logic programs has been proposed, using **CGEs** as a general control construct. Complete forward and backward procedural semantics for such expressions, which preserve the concept of "don't know" non-determinism, were introduced. Also, memory management and goal scheduling issues and their interaction were considered, and a series of mechanisms which can be used to support the algorithms involved at the abstract machine level were described. Finally, a parallel Abstract Machine design was proposed and its performance evaluated. This design is considered to be useful both for the implementation of the execution model on an existing multiprocessor, or as a starting point in the design of a special purpose parallel inference architecture.

It is argued that the following characteristics of the execution model and Abstract Machine meet the criteria initially proposed as the objectives of this research and make the model an attractive vehicle for the implementation of goal independence parallelism:

- *Efficient support of AND-Parallelism and "don't know" non-determinism.* The backward execution semantics ensure support for full "don't know" non-determinism during parallel execution. Variable binding conflicts are detected and dealt with efficiently through the forward execution semantics of the **CGEs**.

- *Extended support for* **WAM** *optimizations.* The parallel machine still

supports last call optimization, environment trimming, unification customization, clause indexing, space retrieval on backtracking and all the other storage and performance optimizations of the **WAM**. In particular, *the total amount of storage needed by the parallel system is essentially equivalent to that of a sequential* **WAM** *implementation.* Storage is still *always retrieved from the top of all stacks* during backtracking thus *simplifying memory management and minimizing the occurrence of garbage collection.*

- *User-transparency of control issues.* Scheduling of processes is done completely at run-time. All communication and synchronization issues are concealed within the semantics of the **CGE**s and can be thus hidden from the user.

- *Distributed control.* The only centralized operation in the system is that of the scheduling network, whose delay can be kept sublinear.

- *Restricted intelligent backtracking.* The backtracking scheme efficiently implements a limited form of intelligent backtracking.

- *"Soft" degradation of performance with resource exhaustion.* Code generated from conditional graph expressions will automatically run sequentially if there are no free processors available (and even then advantage is still taken from the semantics of the **CGE**s for supporting restricted intelligent backtracking with low overhead).

- *Compatibility with conventional* **WAM** *code.* Sequential execution is supported through conventional **WAM** instructions and therefore existing compiler technology can be taken advantage of. This independence of the sequential from the parallel instructions make it possible to extend other similar languages or models to a parallel environment using the same techniques.

- *Sequential speed equivalent to the* **WAM**. Sequential parts of the code or conventional programs which have not been annotated run on one processor at essentially the same speed as in a **WAM** implementation.

- *Efficiency.* A conscious effort has been made throughout the design towards optimizing performance while reducing overhead and minimizing communication, synchronization and storage requirements. This efficiency has been confirmed by the simulations.

It is argued that other solutions previously proposed either burden the user with taking care of control issues which it is considered should be hidden as much as possible, give up concepts which are generally regarded as important in Logic Programing (such as "don't know" non-determinism), or lack the potential for storage efficiency and performance improvement that the **WAM** has brought to the sequential logic programming arena. The fact that the definition of this model has been carried out to a relatively low level has made detailed simulations possible so that a more realistic performance evaluation than that of previously proposed models could be performed. This level of detail also makes the model more amenable to a complete implementation, although some freedom has been left at this level in the definition of the parallel Abstract Machine.

8.1 Areas of Future Research

It is hoped that the research presented herein will only be a beginning: although the model presented is relatively self contained it also leaves many related areas open for future study. Some of these areas are:

- *Automatic generation of* **CGE**'*s*: the "correctness" conditions proposed in Chapter 4 and a data dependency analysis of the source program (perhaps aided by some user annotations) constitute a starting point for the automatic generation of **CGE**'s. Recent work relevant to the detection of determinacy in logic programs [24] can be also applied in order to take advantage of the intrinsic efficiency of determinate parallel calls.

- *Study of the intrinsic Goal Independence parallelism present in typical programs*: such a study would of course be greatly simplified by the existence of the above mentioned tool.

- *Support for OR-parallelism*: The Argonne Labs. model [55] can be used as a starting point for a **WAM**-based implementation of user annotated OR-parallelism. Bound guided OR-parallelism [41] [45] could be very effective in nondeterministic, intensive search applications. The problem of efficiently combining OR- and AND-Parallelism in a practical system is proposed as a subject open for future research.

- *A study of the memory referencing behavior of the system*: this information will be important in the selection of a memory architecture in an eventual implementation of the model.

- *Implementation of the model*: If a hardware implementation is being considered, basic **WAM** designs can be used as a starting point for the processing elements (such as [77] or [28]), but the additional registers and data areas, interrupt and communication capabilities as well as a common access memory system and/or interconnection network have to be added. Alternatively, the model can be implemented on an existing conventional multiprocessor. Several aspects of the Abstract Machine design need to be further refined for a practical implementation.

- *Inclusion of a database interface mechanism*: This interface should be able to support semantic paging, sets, extended indexing, etc. as well as "search parallelism".

- *Support for other languages*: finally, specific additions or modifications can be introduced in the model in order to support other logic and/or functional languages or features not currently supported such as *streams* or *goal suspension*.

Appendix A.

Other Examples of Compiled Code

In this appendix several examples of clauses and their corresponding object code are offered in order to illustrate the function of the different instructions of the Abstract Machine presented in Chapter 7 and to provide guidelines for compiling logic programs annotated with **CGE**s into that instruction set. The emphasis in this appendix is on showing how the instruction set can be used to encode and execute **CGE**s. Thus, little attention is paid to how the expressions were generated from the original clause or whether the expression used is optimal or not. Nevertheless, the expressions are believed to be at least "correct", i.e. they do not generate incorrect parallelism by spawning two goals which are dependent in parallel. Also, in order to pose the emphasis on the "new" instructions, examples which make use of **WAM** instructions whose operation is more or less complicated have been purposedly avoided where possible, specially if the details of their implementation are completely independent of the workings of the parallel model. This includes some unification instructions, the treatment of "unsafe" variables etc. Also for clarity, the order in which parallel goals are pushed in the object code has been left the same as in the textual ordering of goals inside the **CGE**, although, if the same order as the sequential model is to be preserved in the generation of alternatives, and a *Goal* **Stack** is being used, then goals should be pushed in *reverse order* (as illustrated in Chapter 7).

A.1 Checking More Complex Conditions

This example shows how to arrange check instructions in order to test for more complex conditions than the simple "ground(X,Y)" used in the example in Chapter 7. A similar clause is used:

Original Clause:

```
f(X,Y,Z):- a(X,Y,N),
           b(X,Y), c(Z,X), d(X,N),
           e(X,Y,Z,N).
```

Suppose that the following **CGE** was generated for the clause:

Embedded Conditional Graph Expression Form:

```
f(X,Y,Z):- a(X,Y,N),
           (  ( ground(X,Y,Z,N)
                ;
                (ground(X), indep(Y,Z,N))
              )
           | b(X,Y) & c(X,Z) & d(X,N)
           ),
           e(X,Y,Z,N).
```

The **CGE** is interpreted as "b, c, and d can run in parallel if X, Y, Z and N are ground. They can also run in parallel if X is ground and Y, Z, and N are independent. Otherwise they have to run sequentially". Here is the corresponding Abstract Machine code:

Parallel Abstract Machine Code (Compiler Output):

```
procedure  f/3

_277:
        allocate                | Push an environment for f on to the local
                                | stack with space for X, Y, Z, and N.
        ------------------------------------------------------------------------
                                | f(X,Y,Z) :- ...
        get_variable  Y4,A1     | X <- (A1)
```

```
        get_variable  Y3,A2      | Y <- (A2)
        get_variable  Y2,A3      | Z <- (A3)
        ------------------------------------------------------------------------
                                 | ... :- a(X,Y,Z), ...
                                 | Optimization: X and Y are still in A1
                                 | and A2, so they do not need to be
                                 | loaded.
        put_variable  Y1,A3      | (N) -> A3
                                 | A new uninstantiated variable is
                                 | created in Y1 (i.e. "N"), and its
                                 | address is passed in Argument register
                                 | A3.
        call  a/3,4              | call a, 4 permanent variables still needed
        ------------------------------------------------------------------------
                                 | CHECKS:
                                 |
                                 | We first try the first alternative of
                                 | the disjunction:
                                 | ... (ground(X,Y,Z,N) ; ...
                                 |
        check_me_else _351       | This is the address of the other
                                 | alternative in the disjunction.
                                 |
        check_ground Y4          | ground X ? (else _351)
        check_ground Y3          | ground Y ? (else _351)
        check_ground Y2          | ground Z ? (else _351)
        check_ground Y1          | ground N ? (else _351)
        execute _352             | ... all checks succeeded: go to
                                 | parallel code.
        ------------------------------------------------------------------------
_351:                            | The first alternative did not succeed.
                                 | Try the other one:
                                 |
        check_me_else _350       | If we fail now, we go to the sequential code
                                 |
        check_ground Y4          | ... ( ground(X), ...
                                 |
                                 | ... , (indep(Y,Z,N) ) ...
                                 | (all combinations have to be checked:)
        check_independent Y3,Y2  | indep(Y,Z) ?
        check_independent Y3,Y1  | indep(Y,N) ?
        check_independent Y2,Y1  | indep(Z,N) ?
        ------------------------------------------------------------------------
_352:                            | PARALLEL CODE:
        allocate_pcall 3,4       |
        check_ready 1,_351       | ... b(X,Y) & c(Z,X) & d(X,N) ...
        put_value Y4,A1
        put_value Y3,A2
        push_call b/2,1
        check_ready 2,_351
        put_value Y2,A1
        put_value Y4,A2
        push_call c/2,2
        check_ready 3,_351
        put_value Y4,A1
        put_value Y1,A2
        push_call d/3,3
_351:
```

```
        pop_pending_goal
        wait_on_siblings
        execute _355            | ... continue with "e".
        ------------------------------------------------------------------------
_350:                           | SEQUENTIAL CODE:
        put_value  Y4,A1
        put_value  Y3,A2        | ... b(X,Y), c(Z,X), d(X,N) ...
        call  b/2,4
        put_value  Y2,A1
        put_value  Y4,A2
        call  c/2,4
        put_value  Y4,A1
        put_value  Y1,A2
        call  d/2,4
        ------------------------------------------------------------------------
_355:                           | ...  , e(X,Y,Z,N).
        put_value  Y4,A1
        put_value  Y3,A2
        put_value  Y2,A3
        put_value  Y1,A4
        deallocate
        execute  e/4

end
```

The above example shows one way of implementing the original graph expression. Still, a smart compiler could catch the double check for ground(X) present in that expression and generate code which does this check only once:

Parallel Abstract Machine Code (Compiler Output):

```
        .
        .
        .
        check_me_else _350
        check_ground Y4         | ground X ?
        check_me_else _351
        check_ground Y3         | ground Y ?
        check_ground Y2         | ground Z ?
        check_ground Y1         | ground N ?
        execute _352
_351:
        check_me_else _350
        check_independent Y3,Y2 | indep(Y,Z) ?
        check_independent Y3,Y1 | indep(Y,N) ?
        check_independent Y2,Y1 | indep(Z,N) ?
_352:
        ... parallel code
_350:
        ... sequential code
```

This is equivalent to the immediate interpretation of the following conditions:

Embedded Conditional Graph Expression Form:

```
... ground(X), (ground(Y,Z,N);(indep(Y,Z,N)) ...
```

Another possible implementation is:

Embedded Conditional Graph Expression Form:

```
... ground(X), indep(Y,Z,N) ...
```

This would be compiled as:

Parallel Abstract Machine Code (Compiler Output):

```
        .
        .
        .
        check_me_else _350
        check_ground Y4
        check_independent Y3,Y2
        check_independent Y3,Y1
        check_independent Y2,Y1
        ... parallel code
_350:
        ... sequential code
```

This last possibility is more compact but it could be less efficient than the previous one if Y1,Y2 and Y3 (N,Z,Y) are often *ground* since the check for independence obviously involves more overhead.

A.2 All Goals in Parallel: last call optimization issues

Given the following clause and associated CGE:

Original Clause:

```
f(X,Y,Z):- a(X), b(Y), c(Z).
```

Embedded Conditional Graph Expression Form:

```
f(X,Y,Z):- (indep(X,Y,Z) | a(X) & b(Y) & c(Z) ).
```

they can be implemented by the following code:

Parallel Abstract Machine Code (Compiler Output):

```
procedure  f/3

_229:                           | f(X,Y,Z) :-  ...
        allocate
        get_variable  Y2,A2
        get_variable  Y1,A3
        ------------------------------------------------------------------------
        check_me_else _350      | ... (indep(X, Y, Z) | ...
        check_independent A1,Y2
        check_independent A1,Y1
        check_independent Y1,Y2
        ------------------------------------------------------------------------
                                | ... a(X) & b(Y) & c(Z).
        allocate_pcall 3,2
        check_ready 1,_351
        push_call a/2,1         | "a" can now run
        check_ready 2,_351
        put_value Y2,A1
        push_call b/1,2         | "b" can now run
        check_ready 3,_351
        put_value Y1,A1
        push_call c/1,3         | "c" can now run
_351:
        pop_pending_goal
        wait_on_siblings
        deallocate              | has to be after "wait_on_siblings"!
        proceed                 | return to parent.
        ------------------------------------------------------------------------
_350:
        call  a/1,2
        put_value  Y2,A1
        call  b/1,1
```

```
        put_value  Y1,A1
        deallocate
        execute  c/1

end
```

Note the difference with previous examples: the last call in the body of the clause is now inside the parallel expression. If execution is sequential (_350:) the environment can be thrown away as usual before the last call ("`deallocate, execute c/1`": last call optimization). This can be done because in this case we know that all other goals which need access to the permanent variables in the environment (i.e. "a" needing X and "b" needing Y) have already finished their execution. The values needed by "c" are already loaded in the argument registers, so there is no need for the environment any more[60]. The situation is different if execution proceeds in parallel: at the time of executing "c", "a" and "b" might still be unifying their heads using the variables in the local environment, so it cannot be thrown away yet. Instead, after executing all local goals the processor has to wait for completion of the remote ones (i.e. exit from the `wait_on_siblings` instruction) before the environment can be discarded ("`deallocate`"). Note that this does not mean that space is not retrieved any more, just that this retrieval is delayed until completion of the last call. The special case when this can be very costly is when the last goal is a recursive call to the same procedure (tail recursion) or when its descendents in turn contain a call to this procedure (mutual recursion).

In cases where memory space is a problem it might be interesting to give up some parallelism in order to recover last call optimization. This can be easily annotated by imposing sequential ordering for the last goal:

[60]This explanation is necessarily oversimplified since concepts such as "unsafe" variables have not been introduced. It is assumed that the reader who understands the naiveness of this explanation doesn't need it anyway!

Embedded Conditional Graph Expression Form:

```
f(X,Y,Z):- (indep(X,Y) | a(X) & b(Y)), c(Z).
```

which would generate the following code:

Parallel Abstract Machine Code (Compiler Output):

```
procedure f/3

_229:
        allocate
        get_variable Y2,A2      | CHECKS
        get_variable Y1,A3
        check_me_else _350
        check_independent A2,A3
        ------------------------------------------------------------------------
                                | PARALLEL CODE
        allocate_pcall 2,2
        check_ready 1,_351
        push_call a/2,1
        check_ready 2,_351
        put_value Y2,A1
        push_call b/1,2
_351:
        pop_pending_goal
        wait_on_siblings
        execute _352
        ------------------------------------------------------------------------
_350:                           | SEQUENTIAL CODE
        call a/1,2
        put_value Y2,A1
        call b/1,1
        ------------------------------------------------------------------------
_352:                           | LAST CALL
        put_value Y1,A1
        deallocate
        execute c/1

end
```

Note that now last call optimization is fully supported: The environment is thrown away before the last call. If "a" and "b" are determinate (i.e. there is only one alternative clause) the *Parcall Frame* will have all its entries marked as having "no alternatives" and the *Parcall Frame* itself will be discarded upon exit from the "`wait on siblings`" instruction. Of course this "trick" of forcing sequential execution of

the last goal is of no use if the graph expression contains only two goals: there would be only one goal left and thus no parallelism.

A.3 A Nested Parallel Call (using dummy calls)

Only code for non-nested **CGEs** has been generated up to now. The following examples show how nested expressions can be implemented with the same instruction set. Suppose the (nested) **CGE** generated for the following clause

Original Clause:

```
f(X,Y):- g(X,Y), h(X), p(Y).
```

is

Embedded Conditional Graph Expression Form:

```
f(X,Y):- (ground(X,Y) |
            g(X,Y) &
            ( indep(X,Y) | h(X) & k(Y) )
          ).
```

One obvious way of generating code for a nested expression is by substituting expressions inside other expressions by dummy calls to new clauses in which all variables needed are transferred as arguments of the dummy call. In the above case, the compiler would generate an intermediate version of this expression which would include a dummy call (this would be done in one of the early passes, i.e. at "unravel" for Van Roy's [84] compiler):

Embedded Conditional Graph Expression Form:

```
f(X,Y):- (ground(X,Y) |
            g(X,Y) &
            dummy1(X,Y)
```

```
        ).

dummy1(X,Y):- (indep(X,Y) | h(X) & k(Y)).
```

Here is the corresponding output code:

Parallel Abstract Machine Code (Compiler Output):

```
procedure  f/2

_308:
        allocate
        get_variable  Y2,A1      | (X) <- A1
        get_variable  Y1,A2      | (Y) <- A2
        check_me_else _350
        check_ground A1          | ground X ?
        check_ground A2          | ground Y ?
        allocate_pcall 2,2
        check_ready 1,_351       |
        (put_value Y2,A1)        | Not needed: X still in A1
        (put_value Y1,A2)        | Not needed: Y still in A2
        push_call g/2,1
        check_ready 2,_351
        (put_value Y2,A1)        | X still in A1
        (put_value Y1,A2)        | Y still in A2
        push_call dummy/2,2
_351:
        pop_pending_goal
        wait_on_siblings
        deallocate
        proceed
        ------------------------------------------------------------------------
_350:                            | Sequential code.
        put_value  Y2,A1
        put_value  Y1,A2
        call  g/2,2
        put_value  Y2,A1
        put_value  Y1,A2
        deallocate
        execute  dummy/2


procedure  dummy/2

_337:
        allocate
        get_variable  Y1,A2
        check_me_else _360
        check_independent A1,A2
        allocate_pcall 2,1
        check_ready 1,_361
        push_call h/1,1
        check_ready 2,_361
        put_value Y1,A1
```

```
        call k/1,2
_361:
        pop_pending_goal
        wait_on_siblings
        deallocate
        proceed
        ------------------------------------------------------------------------
_360:                           | Sequential code
        call  h/1,1
        put_value  Y1,A1
        deallocate
        execute  k/1

end
```

Note again how many of the instructions are not really needed and an optimizing pass of the compiler can easily detect them. The main advantage of this approach is clearly its simplicity. This means that little modification (other than the addition of a simple source level transformation module) is needed in modifying a conventional compiler in order to support nested CGEs. The main disadvantage with this approach is the overhead in going through an extra procedure call for each level of nesting in the CGEs. In reality this is not as bad as it looks because none of these calls involve any unification, since they are only used for parameter transfer. Thus all of the extra instructions involved are of the "put_value" / "get_variable" types which imply minimal overhead. In addition, most of them will be superfluous (as in the previous example). The other disadvantage is that a simple compiler may generate a new environment for the dummy call that is not really needed: since the new clause is really using the same variables as the calling one the environment can be shared by both (they were really only one clause originally).

In fact, there is no need for a dummy call at all: nested graph expressions can be coded completely in-line. In-line nested parcall coding is obviously more efficient in execution time since it avoids the overhead of going through a goal invocation step for each graph expression. The main disadvantage for in-line code is that due to the lack

of a "subroutine call" in the **WAM** instruction set, parts of the program have to be duplicated.

Nevertheless, there are many advantages to the "dummy call" approach, specially for a first implementation: it is very easy to implement using a conventional compiler by performing a source-level transformation on the input code. Also note that control over parallel calls is actually distributed in the case of a dummy call, because the dummy call itself is transferred as a goal and a remote process can pick it up and take control of the new graph expression. In the in-line approach all control is kept local within the process executing the clause with the graph expression. The "dummy call" approach thus means more parallelism in practice and less memory contention since *Parcall Frames* and copies of the environment are then distributed across processors.

A.4 Other Types of Graph Expressions

This example shows how a smart compiler can take advantage of the instruction set to optimize the code beyond the literal meaning of the **CGE**'s. The same clause as in the previous example will be used:

Original Clause:

```
f(X,Y):- g(X,Y), h(X), k(Y).
```

Embedded Conditional Graph Expression Form:

```
f(X,Y):- (ground(X,Y) |
          g(X,Y) &
          ( indep(X,Y) | h(X) & k(Y) )
         ).
```

In this **CGE** it is obvious that if X and Y are ground then all goals can run

in parallel and there is no need for the check for independence before scheduling "h" and "k". The actual meaning extracted from the expression could be something like

Annotated Code (free syntax):

```
f(X,Y):- ((ground(X,Y) -> g(X,Y) & h(X) & k(Y))
          ;
          (g(X,Y), ((indep(X,Y) -> h(X) & k(Y))
                    ;
                    (h(X), k(Y))
                   )
          )
         ).
```

This can be read as "if X and Y are ground then run "g", "h", and "k" in parallel. Otherwise, run "g" first and then check and see if X and Y are independent. If they are, run them in parallel, otherwise, run them in sequence". This interpretation obviously goes beyond the meaning of the original graph expression and it shows how the instruction set can be used to implement other types of expressions which might overcome some of the shortcomings of CGE's. The above (free syntax) expression could be implemented as follows:

Parallel Abstract Machine Code (Compiler Output):

```
procedure  f/2

_308:
        allocate
        get_variable  Y2,A1
        get_variable  Y1,A2
        ------------------------------------------------------------------------
        check_me_else _350      | see if ground X,Y
        check_ground A1
        check_ground A2
        ------------------------------------------------------------------------
        allocate_pcall 3,2      | if X,Y are ground, run g, h, and k in parallel
        check_ready 1,_351
        (put_value Y2,A1)
        (put_value Y1,A2)
        push_call g/2,1         | g can now run
        check_ready 2,_351
```

```
        (put_value Y2,A1)
        push_call h/1,2         | h can now run
        check_ready 3,_351
        put_value Y1,A1
        push_call k/1,3         | k can now run
_351:
        pop_pending_goal        | i.e. remaining goals from "ground"
        wait_on_siblings
        deallocate
        proceed                 | exit
        ------------------------------------------------------------------------

_350:                           | X and Y are not ground:
        put_value  Y2,A1        | run g first, then check the others
        put_value  Y1,A2
        call  g/2,2             | run g
        ------------------------------------------------------------------------
        check_me_else _380      | see if X,Y independent
        check_independent A1,A2
        ------------------------------------------------------------------------
        allocate_pcall  2,2     | X, Y are independent: run h and k in parallel
        check_ready 1,_381
        (put_value  Y2,A1)
        push_call h/1,1         | h can now run
        check_ready 2,_381
        put_value Y1,A1
        push_call k/1,2         | k can now run
_381:
        pop_pending_goal
        wait_on_siblings
        deallocate
        proceed                 | exit

        ------------------------------------------------------------------------

_380:                           | X, Y are dependent:
        put_value Y2,A1         |run h and k sequentially
        call h/1,1
        put_value Y1,A1
        deallocate
        execute  k/1


end
```

Appendix B.

Benchmarks and Simulation Results

Details about the simulator and the simulations performed are given in this appendix. The "**pwam**" simulator is a direct implementation of the "marker" model described in Chapter 7, including the extensions for dealing with determinate execution. It is based on the Berkeley PLM sequential simulator [27] which is a direct emulation of the **WAM** as described by Warren [88]. The additional instructions and data areas described in Chapter 7, support for the simulation of multiple processors, and some "hooks" for computing total execution time (i.e. speedup, based on simulations at the microcode level) and for computing "work", "wait", and "idle" times and other types of events were added to this simulator by Richard Warren and the author.

B.1 Information Obtained: a Sample Run

In order to illustrate the type of information which was obtained from the simulations, the complete simulator output for a simple two processor run will be shown. The example is the popular symbolic derivation benchmark "deriv.pl". The source (Prolog) code for this benchmark follows:

```
main:- expression(X), d(X, x, Y), write(Y).

d(U+V,X,DU+DV)                  :-  d(U,X,DU), d(V,X,DV).
d(U-V,X,DU-DV)                  :-  d(U,X,DU), d(V,X,DV).
d(U*V,X, DU*V+U*DV)             :-  d(U,X,DU), d(V,X,DV).
d(U/V,X,(DU*V-U*DV)/V^2)        :-  d(U,X,DU), d(V,X,DV).
d(U^N,X, DU*N*U^N1)             :-  integer(N), N1 is N-1, d(U,X,DU).
d(-U,X,-DU)                     :-  d(U,X,DU).
d(exp(U),X,exp(U)*DU)           :-  d(U,X,DU).
d(log(U),X,DU/U)                :-  d(U,X,DU).
d(X,X,1):-!.
d(C,X,0).

expression( 3 * x).
```

The **d** rules are clearly the simple definitions of symbolic derivation. The main program binds X to an expression (simply 3*x in this case), finds the derivative with respect to x, and prints it. This program offers some opportunity for parallelism in the body of the first four clauses (for example, the two derivatives which have to be found in order to generate the derivative of a sum can be done in parallel) if (as is usually the case) **d** is called with a ground first and second argument, and a free variable as the third argument. For simplicity and efficiency we will compile for this case. The **CGE**'s generated would be:

```
main:- expression(X), d(X, x, Y), write(Y).

d(U+V,X,DU+DV)                  :-  (true ! d(U,X,DU) & d(V,X,DV)).
d(U-V,X,DU-DV)                  :-  (true ! d(U,X,DU) & d(V,X,DV)).
d(U*V,X, DU*V+U*DV)             :-  (true ! d(U,X,DU) & d(V,X,DV)).
d(U/V,X,(DU*V-U*DV)/V^2)        :-  (true ! d(U,X,DU) & d(V,X,DV)).
d(U^N,X, DU*N*U^N1)             :-  integer(N), N1 is N-1, d(U,X,DU).
d(-U,X,-DU)                     :-  d(U,X,DU).
d(exp(U),X,exp(U)*DU)           :-  d(U,X,DU).
d(log(U),X,DU/U)                :-  d(U,X,DU).
d(X,X,1):-!.
d(C,X,0).
```

The object code for the above (annotated) program follows (clauses not relevant to this example, such as the rules for "log", "exp", etc. have been taken out for the sake of brevity):

```
procedure  main/0

_1359:
        allocate
        put_variable    Y2,X1
        call  expression/1,2
        put_unsafe_value  Y2,X1
        put_constant  x,X2
        put_variable  Y1,X3
        call  d/3,1
        put_unsafe_value  Y1,X1
        escape  write/1
        deallocate
        proceed


procedure  d/3

        switch_on_term  _1444,_1444,_1445
_1446:
        try_me_else  _1447              % d(U+V, ...
_1448:
        allocate
        get_variable  Y2,X2
        get_structure  +/2,X1
        unify_variable  X1
        unify_variable  Y3
        unify_nil
        get_structure  +/2,X3
        unify_variable  X3
        unify_variable  Y1
        unify_nil
        allocate_det_pcall 3,2
        put_value  Y2,X2
        push_det_call  d/3
        put_value  Y3,X1
        put_value  Y2,X2
        put_value  Y1,X3
        push_det_call  d/3
        pop_pending_goal
        cut_merge
        deallocate
        proceed
_1447:
        retry_me_else  _1449            % d(U-V, ...
_1450:
        allocate
        get_variable  Y2,X2
        get_structure  -/2,X1
        unify_variable  X1
        unify_variable  Y3

      ...

        proceed
_1449:
        retry_me_else  _1451            % d(U*V ...
_1452:
```

```
        allocate
        get_variable  Y2,X2
        get_structure  */2,X1
        unify_variable  X1
        unify_variable  Y3
        unify_nil
        get_structure  +/2,X3
        unify_variable  X4
        unify_variable  X5
        unify_nil
        get_structure  */2,X4
        unify_variable  X3
        unify_unsafe_value  Y3
        unify_nil
        get_structure  */2,X5
        unify_unsafe_value  X1
        unify_variable  Y1
        unify_nil
        allocate_det_pcall 3,2
        put_value  Y2,X2
        push_det_call  d/3
        put_value  Y3,X1
        put_value  Y2,X2
        put_value  Y1,X3
        push_det_call  d/3
        pop_pending_goal
        cut_merge
        deallocate
        proceed
_1451:
        retry_me_else  _1453            % d(U/V ...
_1454:
        allocate
        get_variable  Y2,X2
        get_structure  //2,X1
        unify_variable  X1

      ...

_1461:
        retry_me_else  _1463            % d(X,X, ...
_1464:
        get_value  X1,X2
        get_constant  &1,X3
        allocate
        cut
        deallocate
        proceed
_1463:
        trust_me_else  fail             % d(C,X, ...
_1465:
        get_constant  &0,X3
        proceed
_1444:
        try  _1464
        trust  _1465
_1445:
        try_me_else  _1466
```

```
        switch_on_structure  8,fudge            % Hash table
        log/1    _1462
        exp/1    _1460
        -/1      _1458
        ^/2      _1456
        //2      _1454
        */2      _1452
        -/2      _1450
        +/2      _1448
_1466:
        retry  _1464
        trust  _1465

procedure  expression/1

_235:
        get_structure  */2,X1                   % 3 * x
        unify_constant  &3
        unify_constant  x
        unify_nil
        proceed


end
```

The output from the simulator follows. The number of processors selected is two. The simulator then prints the sizes of the different data areas in each processor, loads the machine code file ("miniparderiv.w"), and assembles it:

```
        Parallel Logic Abstract Machine Simulator
        -----------------------------------------
                (Version 1.2  30 May 86)
                   ["Marker" Model]


Please enter # of processors: 2
Size of Heaps       = 73728
Size of Stacks      = 49152
Size of Trails      = 32768
Size of Goal Stacks = 2500

Loaded file  miniparderiv.w
```

A symbolic listing of the *Code Space* and the procedure and symbol tables can also be obtained. The execution of the program can be traced in a step by step mode, printing all the registers in each step (as in the first instruction below), in

"trace" mode, where only the instructions executed by each processor are printed (as the rest of the instructions below), or in "normal" mode (no output). In any case, total timings are provided at the end of the simulation:

```
debug-> step

*Execution Trace, # of processors = 2

Proc0: P=00000> allocate
P  = 00000001    CP = 00000000   E  = 00014005   PF = 00014000
TR = 00020000    H  = 00002000   HB = 00002000   S  = 00002000
GS = 00000000    CFA= 00000000   B  = 00014000   Type = Choice Point
N  = 00000000    H2 = 00000010   PDL= 00028000   mode = write
AX1 to AX4 = 00000000   00000000        00000000        00000000
AX5 to AX8 = 00000000   00000000        00000000        00000000
Stopped.

debug-> trace

*Execution Trace, # of processors = 2

Proc1: P=-0002> ***idle_free
Proc0: P=00001> put_variable    Y2,X1
Proc1: P=-0002> ***idle_free
Proc0: P=00002> call    expression/1,2
Proc1: P=-0002> ***idle_free
Proc0: P=00209> get_structure   */2,X1
Proc1: P=-0002> ***idle_free
Proc0: P=00210> unify_constant  &3
Proc1: P=-0002> ***idle_free
Proc0: P=00211> unify_constant  x
Proc1: P=-0002> ***idle_free
Proc0: P=00212> unify_nil
Proc1: P=-0002> ***idle_free
Proc0: P=00213> proceed
Proc1: P=-0002> ***idle_free
Proc0: P=00003> put_unsafe_value        Y2,X1
Proc1: P=-0002> ***idle_free
Proc0: P=00004> put_constant    x,X2
Proc1: P=-0002> ***idle_free
Proc0: P=00005> put_variable    Y1,X3
Proc1: P=-0002> ***idle_free
Proc0: P=00006> call    d/3,1
Proc1: P=-0002> ***idle_free
Proc0: P=00011> switch_on_term  _1444,_1444,_1445
Proc1: P=-0002> ***idle_free
Proc0: P=00197> try_me_else     _1466
Proc1: P=-0002> ***idle_free
Proc0: P=00198> switch_on_structure     8,fudge
Proc1: P=-0002> ***idle_free
Proc0: P=00057> allocate
Proc1: P=-0002> ***idle_free
Proc0: P=00058> get_variable    Y2,X2
Proc1: P=-0002> ***idle_free
Proc0: P=00059> get_structure   */2,X1
```

```
Proc1: P=-0002> ***idle_free
Proc0: P=00060> unify_variable  X1
Proc1: P=-0002> ***idle_free
Proc0: P=00061> unify_variable  Y3
Proc1: P=-0002> ***idle_free
Proc0: P=00062> unify_nil
Proc1: P=-0002> ***idle_free
Proc0: P=00063> get_structure   +/2,X3
Proc1: P=-0002> ***idle_free
Proc0: P=00064> unify_variable  X4
Proc1: P=-0002> ***idle_free
Proc0: P=00065> unify_variable  X5
Proc1: P=-0002> ***idle_free
Proc0: P=00066> unify_nil
Proc1: P=-0002> ***idle_free
Proc0: P=00067> get_structure   */2,X4
Proc1: P=-0002> ***idle_free
Proc0: P=00068> unify_variable  X3
Proc1: P=-0002> ***idle_free
Proc0: P=00069> unify_unsafe_value      Y3
Proc1: P=-0002> ***idle_free
Proc0: P=00070> unify_nil
Proc1: P=-0002> ***idle_free
Proc0: P=00071> get_structure   */2,X5
Proc1: P=-0002> ***idle_free
Proc0: P=00072> unify_unsafe_value      X1
Proc1: P=-0002> ***idle_free
Proc0: P=00073> unify_variable  Y1
Proc1: P=-0002> ***idle_free
Proc0: P=00074> unify_nil
Proc1: P=-0002> ***idle_free
Proc0: P=00075> allocate_det_pcall      3,2
Proc1: P=-0002> ***idle_free
Proc0: P=00076> put_value       Y2,X2
Proc1: P=-0002> ***idle_free
Proc0: P=00077> push_det_call   d/3
Proc1: P=-0002> ***idle_free
Proc0: P=00078> put_value       Y3,X1
Proc1: P=00011> switch_on_term  _1444,_1444,_1445
Proc0: P=00079> put_value       Y2,X2
Proc1: P=00195> try     _1464
Proc0: P=00080> put_value       Y1,X3
Proc1: P=00186> get_value       X1,X2 fail!
Proc0: P=00081> push_det_call   d/3
Proc1: P=00196> trust   _1465
Proc0: P=00082> pop_pending_goal
Proc1: P=00193> get_constant    &0,X3
Proc0: P=00011> switch_on_term  _1444,_1444,_1445
Proc1: P=00194> proceed
Proc0: P=00195> try     _1464
Proc1: P=-0002> ***idle_free
Proc0: P=00186> get_value       X1,X2
Proc1: P=-0002> ***idle_free
Proc0: P=00187> get_constant    &1,X3
Proc1: P=-0002> ***idle_free
Proc0: P=00188> allocate
Proc1: P=-0002> ***idle_free
Proc0: P=00189> cut
```

```
Proc1: P=-0002> ***idle_free
Proc0: P=00190> deallocate
Proc1: P=-0002> ***idle_free
Proc0: P=00191> proceed
Proc1: P=-0002> ***idle_free
Proc0: P=00082> pop_pending_goal
Proc1: P=-0002> ***idle_free
Proc0: P=00083> cut_merge
Proc1: P=-0002> ***idle_free
Proc0: P=00084> deallocate
Proc1: P=-0002> ***idle_free
Proc0: P=00085> proceed
Proc1: P=-0002> ***idle_free
Proc0: P=00007> put_unsafe_value        Y1,X1
Proc1: P=-0002> ***idle_free
+/2 */2 (  0 x ) */2 (  3  1 ) Proc0: P=00008> escape   write/1 <escape>
Proc1: P=-0002> ***idle_free
Proc0: P=00009> deallocate
Proc1: P=-0002> ***idle_free

quit-> Proc0: P=00010> proceed

Stopped.
cpu time is 4.740000 sec


 Total execution time  : 628
 number of processors  : 2


TOTAL TIMINGS


             TYPE OF ACTIVITY   CYCLES          Percentage
        time query to solution   629
             total # of cycles   1258
                    total work   730      58.03
                    total wait   37        2.94
                    total idle   290      23.05


Writing file  miniparderiv.data
```

Note in the above how, for such a small example (little intrinsic parallelism), processor 0 (P0) does most of the work. P1 idles looking for work in the system most of the time (`idle_free` pseudo-instruction). As soon as work is available (finding the derivative of $3*x$ is split into finding that of "x", done as a "local goal" in P0, and that of "3", done in P1) it is soon completed and P1 returns to idle again. This accounts for the high percentage of **idle time** (23.5% - all the numbers above

represent totals for all processors). The small amount of **wait time** (2.94%) represents the time that P0 has to wait for P1 to return after executing its goal (including the scheduling overhead). The **total work** represents the total number of cycles spent doing actual work (i.e. executing instructions) and should be equal to the total number of cycles minus wait and idle time. There is often a slight difference if the results given by the simulator are added up since all these numbers are computed from independent sources. The fact that this difference is always less than 1% provides some additional confidence in the soundness of the simulations.

In addition to the data above, and as an option, full instrumentation results for each processor can be obtained:

```
Run-time statistics for  miniparderiv.data running on 2 processors.

TRACE RESULTS FOR PROCESSOR      0


          INSTRUCTION   Count   Percentage
             allocate   3        5.26
   allocate_det_pcall   1        1.75
      allocate_pcall    0        0.00
                call    2        3.51
        check_ground    0        0.00
    check_independent   0        0.00
       check_me_else    0        0.00
         check_ready    0        0.00
                 cut    1        1.75
                cutd    0        0.00
           cut_merge    1        1.75
          deallocate    3        5.26
              escape    1        1.75
             execute    0        0.00
                fail    0        0.00
            get_list    0        0.00 failed 0
       get_structure    5        8.77 failed 0
        get_variable    1        1.75 failed 0
        get_constant    1        1.75 failed 0
           get_value    1        1.75 failed 0
             get_nil    0        0.00 failed 0
                mark    0        0.00
    pop_pending_goal    2        3.51
           push_call    0        0.00
       push_det_call    2        3.51
           put_value    4        7.02
```

put_constant	1	1.75	
proceed	4	7.02	
put_variable	2	3.51	
put_unsafe_value	2	3.51	
put_list	0	0.00	
put_structure	0	0.00	
put_nil	0	0.00	
pause	0	0.00	
quit	0	0.00	
retry_me_else	0	0.00	
retry	0	0.00	
switch_on_term	2	3.51	
switch_on_structure	1	1.75	
switch_on_constant	0	0.00	
try_me_else	1	1.75	
trust_me_else	0	0.00	
try	1	1.75	
trust	0	0.00	
unify_variable	6	10.53	failed 0
unify_cdr	0	0.00	failed 0
unify_value	0	0.00	failed 0
unify_nil	5	8.77	failed 0
unify_constant	2	3.51	failed 0
unify_void	0	0.00	failed 0
unify_unsafe_value	2	3.51	failed 0
wait_on_siblings	0	0.00	
TOTAL	57		

OTHER ACTIONS (parallel)	Count
parcall fails	0
input fails	0
kill pf	0
special failures	0
context switches	0
# of tries to pick goal	0
Noops	0

OTHER ACTIONS (sequential)	Count
failures	0
unifications	3
unify routine	3
bindings	1
escapes	1
memory reads	64
memory writes	105
dereferences	5
binding trails	2
maximum trail	2
maximum stack	60
maximum heap	16
maximum PDL	0

TIMINGS FOR PROCESSOR 0

TYPE OF ACTIVITY	CYCLES	Percentage
processor execution time	629	
processing	586	93.16
waiting	37	5.88
idle	0	0.00

TRACE RESULTS FOR PROCESSOR 1

INSTRUCTION	Count	Percentage	
allocate	0	0.00	
allocate_det_pcall	0	0.00	
allocate_pcall	0	0.00	
call	0	0.00	
check_ground	0	0.00	
check_independent	0	0.00	
check_me_else	0	0.00	
check_ready	0	0.00	
cut	0	0.00	
cutd	0	0.00	
cut_merge	0	0.00	
deallocate	0	0.00	
escape	0	0.00	
execute	0	0.00	
fail	0	0.00	
get_list	0	0.00	failed 0
get_structure	0	0.00	failed 0
get_variable	0	0.00	failed 0
get_constant	1	16.67	failed 0
get_value	1	16.67	failed 1
get_nil	0	0.00	failed 0
mark	0	0.00	
pop_pending_goal	0	0.00	
push_call	0	0.00	
push_det_call	0	0.00	
put_value	0	0.00	
put_constant	0	0.00	
proceed	1	16.67	
put_variable	0	0.00	
put_unsafe_value	0	0.00	
put_list	0	0.00	
put_structure	0	0.00	
put_nil	0	0.00	
pause	0	0.00	
quit	0	0.00	
retry_me_else	0	0.00	
retry	0	0.00	
switch_on_term	1	16.67	
switch_on_structure	0	0.00	
switch_on_constant	0	0.00	

```
                try_me_else   0        0.00
              trust_me_else   0        0.00
                       try   1       16.67
                     trust   1       16.67
             unify_variable   0        0.00 failed 0
                 unify_cdr   0        0.00 failed 0
               unify_value   0        0.00 failed 0
                 unify_nil   0        0.00 failed 0
            unify_constant   0        0.00 failed 0
                unify_void   0        0.00 failed 0
          unify_unsafe_value 0        0.00 failed 435
            wait_on_siblings 0        0.00

                     TOTAL   6


  OTHER ACTIONS (parallel)   Count
            parcall fails    0
              input fails    0
                  kill pf    0
         special failures    0
         context switches    0
 # of tries to pick goal     50
                    Noops    0


OTHER ACTIONS (sequential)   Count
                  failures   1
             unifications    2
             unify routine   2
                 bindings    1
                  escapes    0
            memory  reads    30
            memory writes    37
             dereferences    1
           binding trails    1
            maximum trail    1
            maximum stack    37
             maximum heap    0
              maximum PDL    0

TIMINGS FOR PROCESSOR    1


          TYPE OF ACTIVITY   CYCLES          Percentage
  processor execution time   629
               processing    144      22.89
                  waiting    0         0.00
                     idle    435      69.16
```

TRACE RESULTS FOR ALL PROCESSORS

INSTRUCTION	Count	Percentage	
allocate	3	4.76	
allocate_det_pcall	1	1.59	
allocate_pcall	0	0.00	
call	2	3.17	
check_ground	0	0.00	
check_independent	0	0.00	
check_me_else	0	0.00	
check_ready	0	0.00	
cut	1	1.59	
cutd	0	0.00	
cut_merge	1	1.59	
deallocate	3	4.76	
escape	1	1.59	
execute	0	0.00	
fail	0	0.00	
get_list	0	0.00	failed 0
get_structure	5	7.94	failed 0
get_variable	1	1.59	failed 0
get_constant	2	3.17	failed 0
get_value	2	3.17	failed 1
get_nil	0	0.00	failed 0
mark	0	0.00	
pop_pending_goal	2	3.17	
push_call	0	0.00	
push_det_call	2	3.17	
put_value	4	6.35	
put_constant	1	1.59	
proceed	5	7.94	
put_variable	2	3.17	
put_unsafe_value	2	3.17	
put_list	0	0.00	
put_structure	0	0.00	
put_nil	0	0.00	
pause	0	0.00	
quit	0	0.00	
retry_me_else	0	0.00	
retry	0	0.00	
switch_on_term	3	4.76	
switch_on_structure	1	1.59	
switch_on_constant	0	0.00	
try_me_else	1	1.59	
trust_me_else	0	0.00	
try	2	3.17	
trust	1	1.59	
unify_variable	6	9.52	failed 0
unify_cdr	0	0.00	failed 0
unify_value	0	0.00	failed 0
unify_nil	5	7.94	failed 0
unify_constant	2	3.17	failed 0
unify_void	0	0.00	failed 0
unify_unsafe_value	2	3.17	failed 290
wait_on_siblings	0	0.00	

```
                    TOTAL   63


      OTHER ACTIONS (parallel)   Count
                 parcall fails   0
                   input fails   0
                       kill pf   0
              special failures   0
              context switches   0
     # of tries to pick goal     50
                         Noops   0


    OTHER ACTIONS (sequential)   Count
                       failures  1
                  unifications   5
                 unify routine   5
                      bindings   2
                       escapes   1
                  memory  reads  94
                  memory writes  142
                   dereferences  6
                 binding trails  3

TOTAL TIMINGS


            TYPE OF ACTIVITY   CYCLES          Percentage
      time query to solution   629
           total # of cycles   1258
                  total work   730     58.03
                  total wait   37       2.94
                  total idle   290     23.05
```

These numbers can be used to show how the overhead in the system due to the control of parallelism is low (for example, in the example above, the number of "new" instructions executed only represents $\sim 7\%$ of the total instruction count, and even less in number of cycles). It is also clear how P1 accounts for all the idle time, while P0 is responsible for all the wait time in the system. Although much of this is rather obvious it is pointed out as an added "confidence factor" for the simulations.

B.2 Efficiency Tests: Synthetic Benchmarks

One of the problems in determining the **efficiency** of the model is obtaining good benchmarks where the amount of available parallelism is known "a priori", so that intrinsic overheads can be discerned from lack of parallelism in the benchmark. The following skeleton will be used for generating a series of programs which represent problems which are symmetric and have fixed intrinsic parallelism (such as, for example, matrix multiplication) from the point of view of a "goal independence" model (`timings.pl`):

```
check:- times(X), p(X), p(X), p(X), p(X), ..., p(X), p(X), p(X).

p(0).
p(X):- Y is (X-1), p(Y).

times(...).
```

Procedure **p** is simply a loop which is executed as many times as indicated by the value of a constant which is placed in the fact `times(...)`. It represents an independent segment of the program which can be executed in parallel with the other calls to **p** in the body of **check**. A series of programs with names "`partimings`*n*`.pl`", where *n* is the number of calls to **p** in the body of **times**, were constructed. Note that *n* represents the amount of intrinsic parallelism in the program. The first element of the series is `timings.pl`:

```
check:- times(X), p(X).

p(0).
p(X):- Y is (X-1), p(Y).

times(128).
```

And this would be the source code for `partimings4.pl`:

```
check:- times(X), ( true | p(X) & p(X) & p(X) & p(X) ).

p(0).
p(X):- Y is (X-1), p(Y).

times(32).
```

Note that although the amount of *intrinsic parallelism* in this program is 4, the amount of *work* is essentially the same as in `partimings.pl`, since each `p` does a quarter of the work. In fact, all members of the series represent the same amount of work.

The above programs are compiled as in the previous examples. The basic sequential problem `timings.pl` compiles to:

```
procedure  check/0

_507:
        allocate
        put_variable  Y1,X1
        call  times/1,1
        put_value  Y1,X1
        call  p/1,1
        put_value  Y1,X1
        call  p/1,1
        put_value  Y1,X1
        call  p/1,1
        put_value  Y1,X1
        call  p/1,1
        ...
        put_value  Y1,X1
        call  p/1,1
        put_value  Y1,X1
        call  p/1,1
        put_unsafe_value  Y1,X1
        deallocate
        execute  p/1


procedure  p/1

        switch_on_term  _546,_547,_547
_546:
        try_me_else  _548
```

```
_549:
        get_constant  &0,X1
        proceed
_548:
        trust_me_else  fail
_547:
        put_structure  -/2,X2
        unify_unsafe_value  X1
        unify_constant  &1
        unify_nil
        allocate
        put_variable  Y1,X1
        escape  is/2
        put_unsafe_value  Y1,X1
        deallocate
        execute  p/1


procedure  times/1

_578:
        get_constant    &...,X1
        proceed


end
```

The parallel versions differ only in the **check** clause. For example, here is the machine code for this clause in **partimings4.w**:

```
procedure  check/0

_507:
        allocate
        put_variable  Y1,X1
        call  times/1,1
        allocate_det_pcall  1,3
        put_value  Y1,X1
        push_det_call  p/1
        push_det_call  p/1
        push_det_call  p/1
        call    p/1,1
        pop_pending_goal
        cut_merge
        deallocate
        proceed
```

Figures B-1 and B-2 show some of the data collected during the simulations

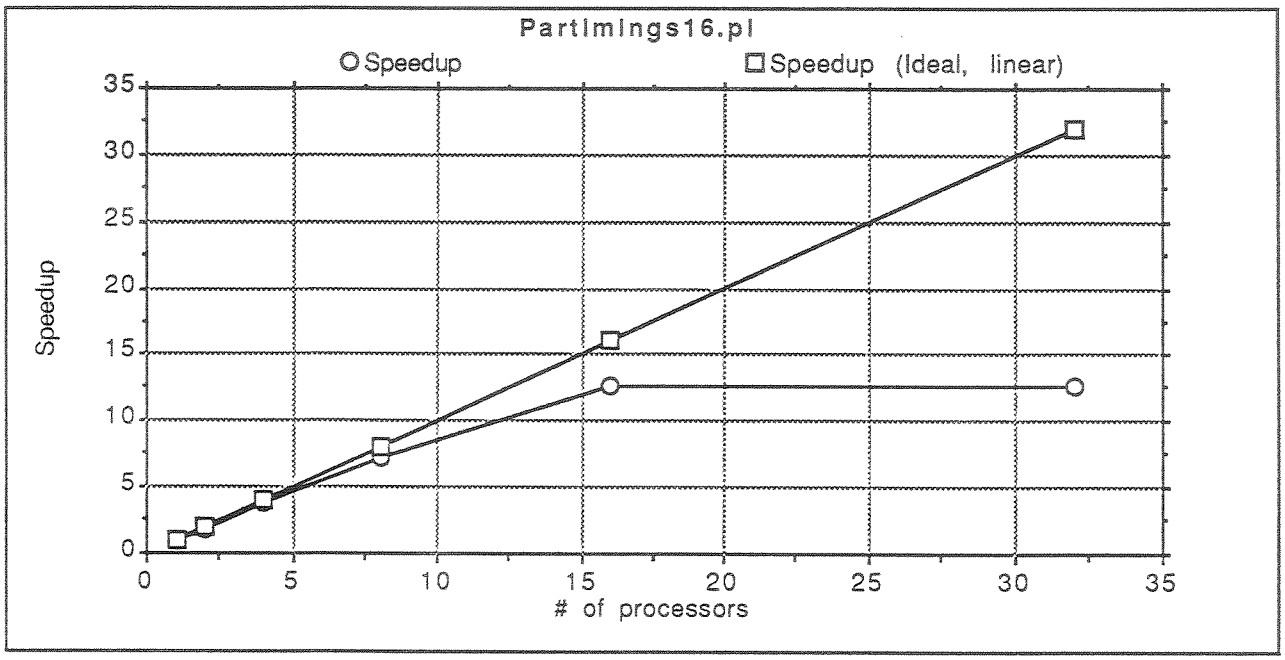

Figure B-1: Speedup vs. # of processors for `partimings16.pl`

for `partimings16.pl`. Figure B-1 plots the speedup in the execution time of `partimings16.pl` as a function of the number of processors. The speedup is linear and close to ideal until the problem runs out of parallelism (for example, for `partimings16.pl`, if more than 16 processors are used). Figure B-2 shows the total amount of work and the total time spent waiting and idling for the same problem (in number of memory cycles), also as a function of the number of processors involved. The total amount of work involved in running the same program sequentially is also plotted in order to estimate the overhead in the model. Note how wait and idle times are very low because of the intrinsic parallelism and symmetry in the problem, and that the efficiency of the model is very high (the amount of parallel work is always less than 10% larger that the sequential work even when 32 processors are working on the problem). Of course, when the system "runs out" of intrinsic parallelism in the problem (such as running `partimings16.pl` on 64 processors) the idle times increase considerably.

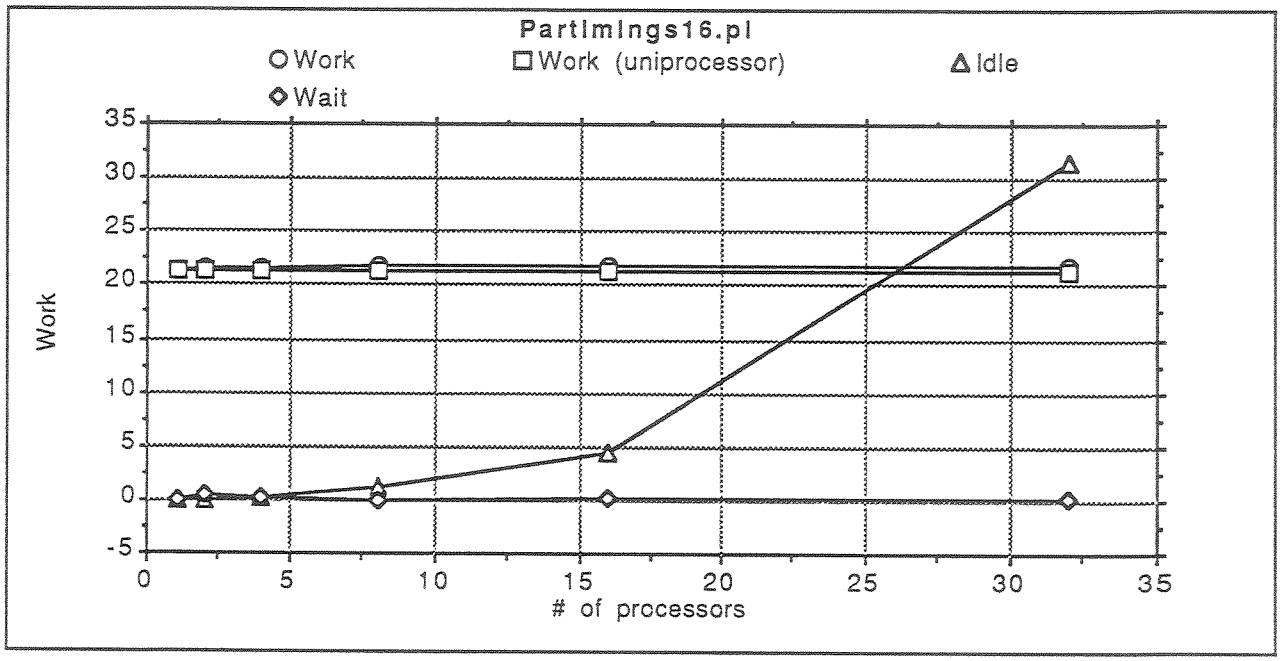

Figure B-2: Wait, Work, and Idle times for `partimings16.pl`

B.3 A More Realistic Problem: Symbolic Derivation

As an example of some more realistic benchmarks which have been run we will return to the symbolic derivation problem, but this time with a much more substantial expression. The basic program is the same as before, but the expression is now:

$$exp + exp - exp\ exp\ /\ exp\ exp\ /\ exp$$

where

$$exp = \frac{3x + 4e^{x^3}\ logx^2 - 2}{-3x + \frac{5}{e^{x^4} + 2}}$$

Or, in Prolog:

```
expression( Exp
          + Exp
          - Exp
          * Exp
          / Exp
          * Exp
          / Exp
          ) :- value(Exp).

value(((3*x + (4*exp(x^3)*log(x^2)) -2) /
             ( -(3*x) + 5/(exp(x^4)+2)))).
```

The expression is constructed in several "parts" in order to generate a complicated expression while saving compilation time and keeping the machine code short. Some of the results of the simulations on this example are shown in figures B-3 and B-4. Figure B-5 represents the percentile proportions between the magnitudes shown in figure B-4.

Some comments on these results. Clearly, on a more realistic problem like this one, the wait and idle times are very useful in analyzing the characteristics of the problem. For example, the idle time remains low until there are more than eight processors in the system (idle time less than 10%), and from then on it starts being

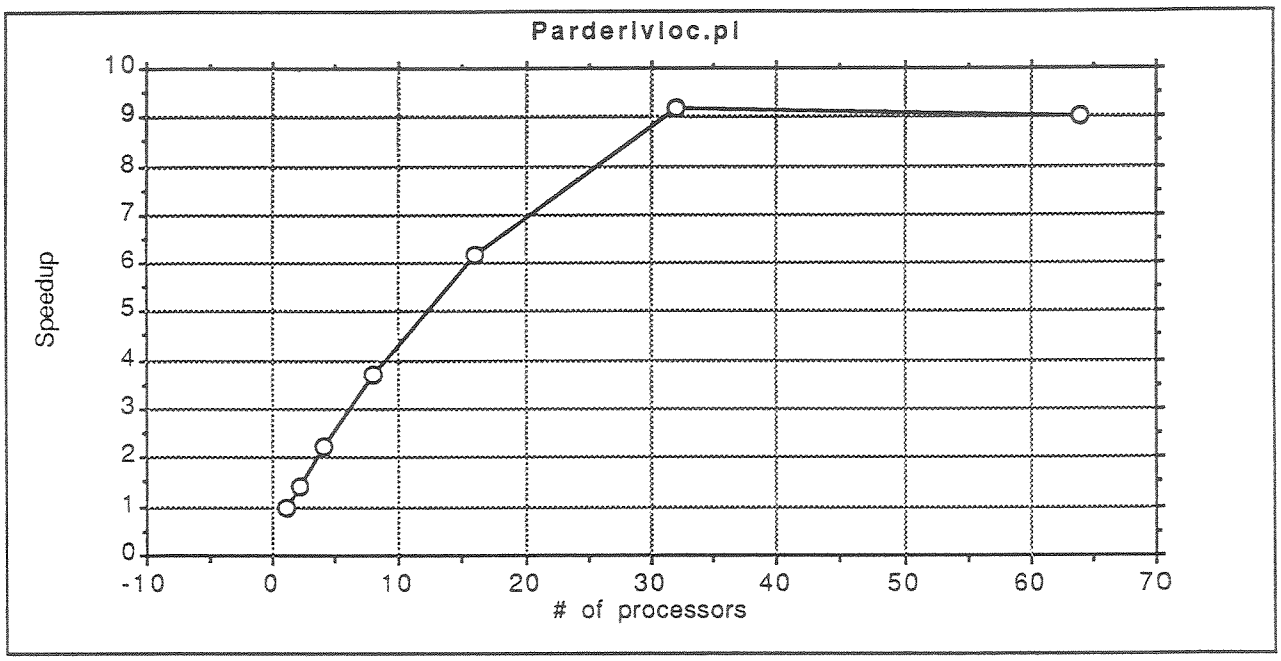

Figure B-3: Speedup vs. # of processors for `parderivloc.pl`

considerable higher. This is indicative of the amount of parallelism in the problem which can be exploited efficiently. However, performance improves steadily until more than 32 processors are used. This provides a measure of the amount of total intrinsic parallelism available. If performance is the key factor, this is the maximum response speed obtainable, at the cost of low processor utilization. This maximum speedup is in the order of 10 for this small and not "particularly parallel" problem.

Note that this derivation example represents an extreme case of "fine grain" parallelism for a *Goal Independence* model: whereas in the `partimings` example (which basically models a "matrix multiplication" problem) there was considerable sequential work involved for each goal spawned, in the derivation example almost every goal invocation involves process spawning, and the work done for each spawn is limited to a few unification instructions. This fine grain accounts for the higher

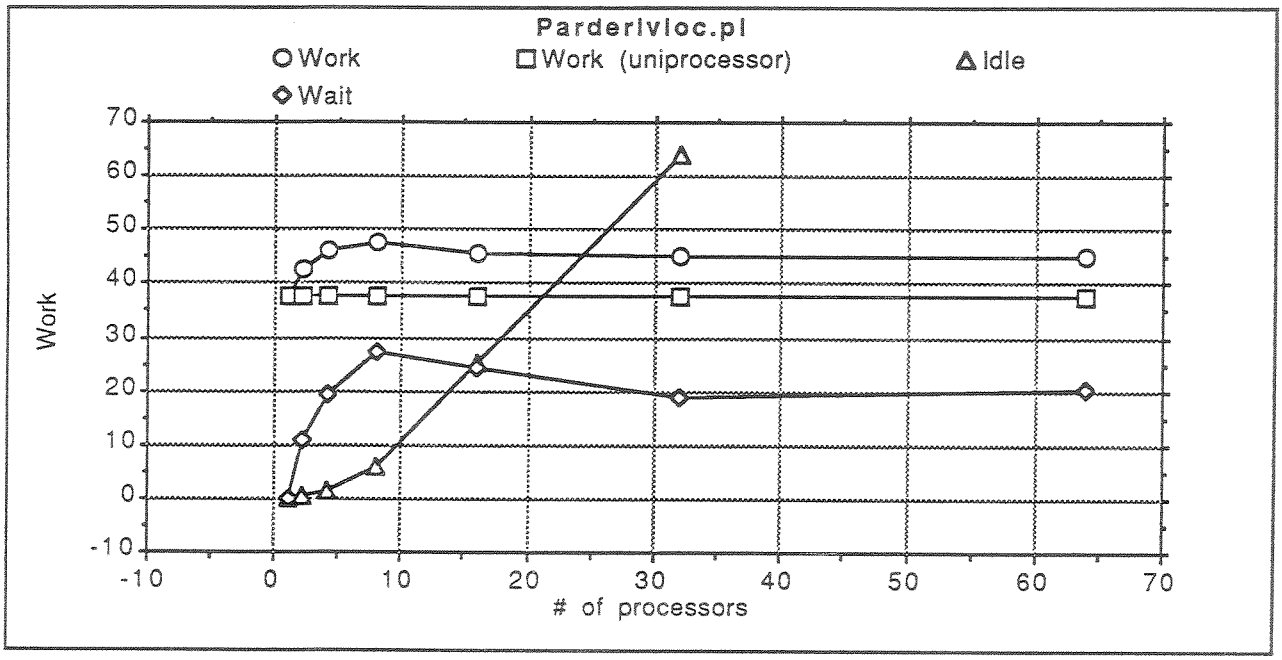

Figure B-4: Wait, Work, and Idle times for `parderivloc.pl`

overhead incurred into (typically 14%, difference between sequential and parallel work) when compared to the previous example. However, this overhead is still low when compared to other models, specially considering that it represents an extreme case, and that effective speedup can still be obtained as pointed out above.

Wait times have their origin not in the lack of intrinsic parallelism in the problem, but in the asymmetry of the tasks executed by the different processors: if the parent processor finishes with its local goals, then it may have to wait for a sibling which may have picked up a more complicated goal to return (the "join"). Of course, these wait times can be eliminated by introducing multiprocessing *if there is available parallel work in the system*: if a processor has to wait, it can start a new process which will pick up some of that work. However, if performance is the key factor, then this work should be taken care of by adding more processors: since these processors

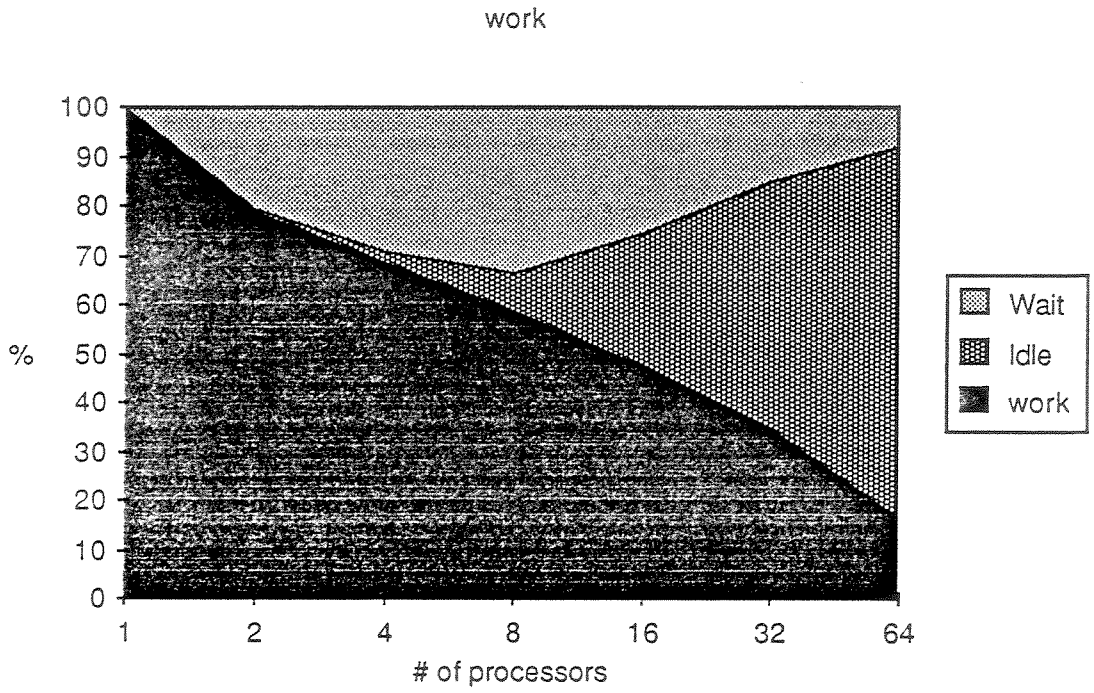

Figure B-5: Wait, Work, and Idle times for `parderivloc.pl` (%)

would not have to switch contexts, the final performance would be higher. Also, if the children of a processor all return while it is executing another process, it will have to time-slice two (or eventually more) processes, with a net loss in performance. Therefore, a new process in a waiting processor should only be started if there are no idle processors in the system. An alternate way of eliminating "wait" times (the "continuation" method) was sketched in Chapter 7.

B.4 Megalips Now?

Simply as a matter of exercise, since so many other factors, such as the number of processors in the system and the size and amount of intrinsic parallelism in the problem can dramatically affect performance figures, an estimate will be attempted of the potential for performance of the model for simple cases such as the derivation example shown above. It has been already pointed out how a net speedup

factor of an order of magnitude can be obtained for a relatively small problem with medium intrinsic parallelism. An estimate of the sequential speed of a hardware implementation of the WAM architecture (such as the design proposed by Tick and Warren [77]) could be placed around the 500Klip region (logical inferences per second). Thus, the performance potential of an eventual implementation of the model, for a case such as the derivation example above, can be rated at around 5 Mlips. Of course, this figure can be much higher for problems which exhibit more parallelism and whose grains of computation are larger, such as those modelled by the `"partimings"` examples of previous sections.

B.5 Conclusions and Suggestions for Further Work

The simulations performed were not intended to be an exhaustive study of the model. This task is left as a subject for future research. Their objective was to prove the model's general functionality and viability. The main conclusion from the data offered in the preceding sections is that in addition to functionality, the model exhibits very good efficiency: if the problem has intrinsic goal independence parallelism, the model will take advantage of it with very little overhead. Further simulations on larger and more realistic problems, which are outside the capabilities of the simulator in its present form, should be performed in order to evaluate the amount of intrinsic goal independence parallelism present in such problems. Also, the simulator does not take into account memory reference behavior [75] and contention or interconnection network delay (other than a fixed value), since a precise memory architecture is not specified in the model, but these factors should clearly be considered for an implementation. This is also suggested as a topic for future research (see Chapter 8).

Bibliography

[1] Arthur C. Clarke.
2001, A Space Odyssey

[2] Arvind, and Robert A. Ianucci.
A Critique of Multiprocessing von Neumann Style.
In *Proceedings of the 1983 ACM Conference on Computer Architecture*, pages 426-437. ACM Press, 1983.

[3] J. Backus.
Can Programming be Liberated from the von Neumann Style?
Communications of the ACM 21(8):613-641, 1978.

[4] F. Bacilhon and R. Ramakrishnan.
An Amateur's Introduction to Recursive Query Processing Strategies.
MCC Technical Report DB-091-86, Microelectronics and Computer Technology Corp., 1986.

[5] K. E. Batcher.
Sorting Networks and Their Application.
AFIPS Conf. Proc. 32:307-314, 1968.

[6] M. Bergmann and H. Kanoui.
Application of Mechanical Theorem Proving to Symbolic Calculus.
Third International Symposium on Advanced Computer Methods in Theoretical Physics , June, 1973.
Marseilles C.N.R.S.

[7] P. Borgwardt.
Parallel Prolog Using Stack Segments on Shared Memory Multiprocessors .
In *1984 International Symposium on Logic Programming, Atlantic City,* pages 2-12. IEEE Computer Society Press, Silver Spring, MD, February, 1984.

[8] F. W. Burton and M. R. Sleep.
Executing Functional Programs on a Virtual Tree of Processors.
In *Functional Programming Languages and Computer Architecture*, pages 187-195. October, 1981.

[9] J.-H. Chang, A. M. Despain, and D. DeGroot.
AND-parallelism of Logic Programs Based on Static Data Dependency Analysis.
In *Digest of Papers of COMPCON Spring '85*, pages 218-225. 1985.

[10] A. Church.
The Calculi of Lambda Conversion.
In *Annals of Mathematics Studies*. Princeton University Press, 1941.

[11] A. Ciepilewski and S. Haridi.
Control of Activities in the Or-Parallel Token Machine.
In *1984 International Symposium on Logic Programming, Atlantic City*, pages 49-58. IEEE Computer Society Press, Silver Spring, MD, February, 1984.

[12] A. Ciepielewski.
Towards a Computer Architecture for OR-Parallel Execution of Logic Programs.
PhD thesis, Dept. of Computer Systems, Royal Institute of Technology, 1984.
May 17, 1984.

[13] K. Clark and S. Gregory.
PARLOG: A Parallel Logic Programming Language.
Research Report DOC 83/5, Dept. of Computing, Imperial College of Science and Technology, May, 1983.
University of London.

[14] Clark, K.L. and G. McCabe.
The Control Facilities of IC-Prolog.
Expert Systems in the Micro Electronic Age.
Edinburgh University Press, 1979.

[15] K. L. Clarke, and F. G. McCabe.
micro-Prolog: Programming in Logic.
Prentice-Hall, Englewood Cliffs, N.J., 1984.

[16] W.F. Clocksin and C.S. Mellish.
Programming in Prolog.
Springer-Verlag, Berlin, New York, 1981.

[17] H. Coelho, J. C. Cotta, and L. M. Pereira.
How to Solve it with Prolog.
Ministerio da Habitacao e Obras Publicas, Laboratorio de Engenharia Civil, Lisbon, 1980.

[18] Michael Codish and Ehud Shapiro.
Compiling OR-Parallelism into AND-Parallelism.
In *Proceedings of the Third International Conference on Logic Programming*, pages 283-298. Springer-Verlag, 1986.

[19] J.S. Conery and D.F. Kibler.
Parallel Query Processing in Logic Databases.
Proc. 8th. IJCAI , 1983.

[20] J.S. Conery.
The AND/OR Process Model for Parallel Interpretation of Logic Programs.
PhD thesis, The University of California at Irvine, 1983.
Technical Report 204.

[21] W. D. Hillis.
The Connection Machine.
The MIT Press, Cambridge, Massachusetts, 1986.

[22] *CRAY-1 Computer System Hardware Reference Manual*
Cray Research Inc., Bloomington, Minn., 1977.
Pub. # 2240004.

[23] J. Darlington, A. J. Field, and H. Pull.
The Unification of Functional and Logic Languages.
In D. DeGroot and G. Lindstrom (editors), *Logic Programming: Relations, Functions, and Equations.* Prentice-Hall, Englewood Cliffs, N. J., 1985.

[24] Saumya K. Debray and D. S. Warren.
Automatic Mode Inference for Prolog Programs.
In *1986 Symposium on Logic Programming.* IEEE Computer Society, 1986.

[25] Doug DeGroot.
Restricted And-Parallelism.
Int'l Conf. on Fifth Generation Computer Systems , November, 1984.

[26] T. P. Dobry, A. M. Despain, and Y. N. Patt.
Performance Studies of a Prolog Machine Architecture.
In *Proceedings of the 12 Int'l. Symp. on Computer Architecture*, pages 180-191. IEEE Computer Society Press, 1985.

[27] T. P. Dobry.
PLM Simulator Reference Manual
Computer Science Division, University of California, Berkeley, 1984.

[28] T. P. Dobry, Y. N. Patt, and A. M. Despain.
Design Decision Influencing the Microarchitecture for a Prolog Machine.
In *MICRO 17 Proceedings*. 1984.

[29] John Gabriel, Tim Lindholm, E. L. Lusk, and R. A. Overbeek.
A Tutorial on the Warren Abstract Machine.
Technical Report, Argonne National Laboratory, Argonne, Ill. 60439, 1985.

[30] Steven Gregory.
Design, Application and Implementation of a Parallel Logic Programming Language.
PhD thesis, Imperial College of Science and Technology, 1985.

[31] Manuel V. Hermenegildo.
An Abstract Machine for Restricted AND-parallel Execution of Logic Programs.
In *Proceedings of the Third International Conference on Logic Programming*, pages 25-40. Springer-Verlag, 1986.

[32] Manuel V. Hermenegildo and Roger I. Nasr.
Efficient Implementation of Backtracking in AND-parallelism.
In *Proceedings of the Third International Conference on Logic Programming*, pages 40-55. Springer-Verlag, 1986.

[33] C. A. R. Hoare.
Algorithm64.
Communications of the CM 4:321, 1961.

[34] C. J. Hogger.
Introduction to Logic Programming.
Academic Press, London, 1984.

[35] Hassan Ait-Kaci, Robert Boyer, Roger Nasr.
An Encoding Technique for the Efficient Implementation of Type Inheritance.
Technical Report AI-109-85, Microelectronics and Computer Technology Corporation, 9430 Research Boulevard, Austin, TX 78759, 1985.

[36] G. Kahn and D. B. MacQueen.
Coroutines and Networks of Parallel Processes.
In B. Gilchrist (editor), *Information Processing 77; Proceedings of the IFIP Congress 77*, pages 993-998. Elsevier North-Holland, Amsterdam, 1977.

[37] S. Kasif, M. Kohli, and J. Minker.
PRISM: A parallel inference system for problem solving.
In *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pages 544-546. August, 1983.

[38] Robert M. Keller, Frank C. H. Lin, and Jiro Tanaka.
Rediflow Multiprocessing.
In *Digest of Papers, Spring COMPCON '84*, pages 410-417. IEE Computer Society, 1984.

[39] Kowalski, R.A.
Predicate Logic as a Programming Language.
Proc. IFIPS 74 , 1974.

[40] Robert A. Kowalski.
Logic for Problem Solving.
Elsevier North-Holland Inc., 1979.

[41] G. J. Lipovski and Manuel V. Hermenegildo.
B-LOG: A Branch and Bound Methodology for the Parallel Execution of Logic Programs.
In *Proceedings of the 1985 International Conference on Parallel Processing.* IEEE Computer Society, 1985.

[42] Yow-Jian Lin, Vipin Kumar, and Clemment Leung.
An Intelligent Backtracking Algorithm for Parallel Execution of Logic Programs.
In *Proceedings of the Third International Conference on Logic Programming*, pages 55-69. Springer-Verlag, 1986.

[43] G. Lindstrom and P. Panangden.
Stream-Based Execution of Logic Programming.
In *1984 International Symposium on Logic Programming, Atlantic City*, pages 168-177. IEEE Computer Society Press, Silver Spring, MD, February, 1984.

[44] G. J. Lipovski and A. Tripathi.
A Reconfigurable Varistructured Array Processor.
In *Int. Conf. on Parallel Processing*, pages 165-174. August, 1977.

[45] B. W. Wah, G. J. Li, and C. F. Yu.
The Status of Manip: A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems.
In *Proc. of the 11th. Annual Int'l Symp. on Computer Architecture*, pages 56-64. IEEE Computer Society, 1984.

[46] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M.I. Levin.
LISP 1.5 Programmer's Manual.
MIT Press, Cambridge, MA, 1965.

[47] C. S. Mellish.
An alternative to structure sharing in the implementation of a Prolog interpreter.
Research Paper 150, Department of AI, University of Edinburgh, August, 1980.

[48] C. S. Mellish.
The Automatic Generation of Mode Declarations for Prolog Programs.
DAI Research Paper 163, Department of Artificial Intelligence, Univ. of Edinburgh, 1981.

[49] C.S. Mellish.
Abstract Iterpretation of Prolog Programs.
In *Proceedings of the Third International Conference on Logic Programming*, pages 463-475. Springer-Verlag, 1986.

[50] L. Monteiro.
A Proposal for Distributed Programming in Logic.
Implementations of Prolog.
Ellis Horwood, 1984.

[51] L. Naish.
Automatic Generation of Control for Logic Programs.
Technical Report 83/6, Department of Computer Science, University of Melbourne, 1983.

[52] L. Naish.
All Solutions Predicates in Prolog.
In *1985 Symposium on Logic Programming*, pages 73-78. IEEE Computer Society, 1985.

[53] H. Nishikawa, A. Yamamoto, K. Taki, and S. Uchida.
The Personal Inference Machine (PSI): Its design philosophy and Machine Architecture.
In Universidade Nova de Lisboa (editor), *Logic Programming Workshop '83*, pages 53-73. June, 1983.

[54] *The Architecture of Norma*
Austin Research Center, Burroughs Corp., Austin, TX, 1985.

[55] R. A. Overbeek, J. Gabriel, T. Lindholm, and E. L. Lusk.
Prolog on Multiprocessors.
Technical Report, Argonne National Laboratory, Argonne, Ill. 60439, 1985.

[56] Luis Moniz Pereira, Luis Monteiro, Jose Cunha & Joaquim N. Aparicio.
Delta Prolog: a distributed backtracking extension with events.
In *Proceedings of the Third International Conference on Logic Programming*, pages 69-84. Springer-Verlag, 1986.

[57] Luis M. Pereira and Roger I. Nasr.
Delta-Prolog: A Distributed Logic Programming Language.
In *Proceedings of the Intl. Conf. on 5th. Gen. Computer Systems.* 1984. Japan.

[58] Pereira, L.M., F. C. N. Pereira, and D. H. D. Warren.
User's Guide to DECsystem-10 Prolog
Dept. of Artificial Intelligence, Univ. of Edinburgh, 1978.

[59] Pereira, L.P., and A. Porto.
Intelligent Backtracking and Sidetracking in Horn Clause Programs - the Theory.
Report 2/79, Departamento de Informatica, Universidade Nova de Lisboa, October, 1979.

[60] L. M. Pereira and A. Porto.
Selective Backtracking.
Logic Programming.
Academic Press, 1982, pages 107-114.

[61] A. Porto and M. Filgueiras.
Natural Language Semantics-: A Logic Programming Approach.
In *1984 International Symposium on Logic Programming, Atlantic City*, pages 228-233. IEEE Computer Society Press, Silver Spring, MD, February, 1984.

[62] G. H. Pollard.
Parallel Execution of Horn Clause Programs.
PhD thesis, Imperial College, 1981.
Dept. of Computing.

[63] A. Colmenauer et al.
Prolog II: Reference Manual and Theoretical Model
Groupe d'Intelligence Artificielle, Faculte des Sciences de Luminy, Marseilles, 1982.

[64] *Quintus Prolog Reference Manual*
4 edition, Quintus Computer Systems, 2345 Yale St., Palo Alto, CA 94306, 1984.

[65] G. Radin.
The 801 Minicomputer.
IBM Journal of Research and Development 27(3):237-246, 1983.

[66] U.S. Reddy.
Transformation of Logic Programs into Functional Programs.
In *1984 International Symposium on Logic Programming, Atlantic City*, pages 187-198. IEEE Computer Society Press, Silver Spring, MD, February, 1984.

[67] D. A. Patterson and C. H. Sequin.
RISC I: A Reduced Instruction Set VLSI Computer.
In *Proceedings of the 8th. Int'l Symp. on Computer Architecture*. IEEE Computer Society, 1981.

[68] J. A. Robinson.
A machine oriented logic based on the resolution principle.
Journal of the ACM 12(23):23-41, January, 1965.

[69] Roussel, P.
Prolog: Manuel de Reference et d'Utilisation.
Technical Report, Univ. d'Aix-Marseille, Groupe de IA, 1975.

[70] M. C. Sejnowski, E. T. Upchurch, R. Kapur, D. P. S. Charlu and G. J. Lipovski.
An Overview of the Texas Reconfigurable Array Processor.
In Don Medley (editor), *AFIPS Conference Proceedings, National Computer Conference 1980*, pages 631-643. AFIPS Press, Arlington, Va, May 19-22, 1980.

[71] *BALANCE 8000 Guide to Parallel Programming*
2.0 edition, Sequent Computer Systems, Inc., 1985.

[72] E. Y. Shapiro.
A subset of Concurrent Prolog and its interpreter.
Technical Report TR-003, ICOT, January, 1983.
Tokyo.

[73] *Symbolics 3600 Technical Summary*
Symbolics Inc., Cambridge, Massachusetts, 1983.

[74] N. Tamura and Y. Kaneda.
Implementing Parallel Prolog on a Multiprocessor Machine.
In *1984 International Symposium on Logic Programming, Atlantic City*, pages 42-49. IEEE Computer Society Press, Silver Spring, MD, February, 1984.

[75] Evan Tick.
Prolog Memory-Referencing Behavior.
Technical Report 85-281, Computer Systems Laboratory, 1985.
Stanford University.

[76] Evan Tick.
Lisp and Prolog Memory Performance.
Technical Report 86-291, Computer Systems Laboratory, 1986.
Stanford University.

[77] E. Tick and D.H.D. Warren.
Towards a Pipelined Prolog Processor.
In *1984 International Symposium on Logic Programming, Atlantic City,* pages 29-42. IEEE Computer Society Press, Silver Spring, MD, February, 1984.

[78] C. Whitby Strevens.
The Transputer.
In *Proceedings of the 12th. Int'l Symp. on Computer Architecture,* pages 292-302. IEEE Computer Society, 1985.
Boston, Massachussetts.

[79] Philip C. Treleaven.
The New Generation of Computer Architecture.
Proceedings of the 1983 ACM Conference on Computer Architecture :402-409, 1983.

[80] Sunichi Uchida.
Towards a New Generation Computer Architecture: Research and Development Plan for Computer Architecture in the Fifth Generation Computer Project.
Technical Report TR-001, ICOT-Institute for New Generation Computer Technology, July, 1982.

[81] Ueda, K.
Guarded Horn Clauses.
Technical Report TR-103, ICOT, 1985.
Tokyo.

[82] Kazunori Ueda.
Making Exhaustive Search Programs Deterministic.
In *Proceedings of the Third International Conference on Logic Programming,* pages 270-283. Springer-Verlag, 1986.

[83] M. H. VanEmden, and R. A. Kowalski.
The Semantics of Predicate Logic as a Programming Language.
Journal of the ACM 23:733-742, 1976.

[84] P. Van Roy.
A Prolog Compiler for the PLM.
Master's thesis, University of California at Berkeley, 1984.

[85] David H. D. Warren.
Implementing Prolog - Compiling Predicate Logic Programs.
Technical Report 39 & 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

[86] David H. D. Warren.
Logic for Compiler Writing.
Software Practice and Experience.
1980, pages 97-125.

[87] David H. D. Warren.
An improved Prolog implementation which optimises tail recursion.
DAI Research Report 141, University of Edinburgh, 1980.

[88] David H. D. Warren.
An Abstract Prolog Instruction Set.
Technical Note 309, SRI International, AI Center, Computer Science and Technology Division, 1983.

[89] D.S. Warren.
Efficient Prolog Memory Management for Flexible Control Strategies.
In *1984 International Symposium on Logic Programming, Atlantic City,* pages 198-203. IEEE Computer Society Press, Silver Spring, MD, February, 1984.