Efficient Management of Backtracking
in AND-Parallelism

M. V. Hermenegildo
R. I. Nasr

Abstract

A backtracking algorithm for AND-Parallelism and its implementation at the Abstract Machine level are presented: first, a class of AND-Parallelism models based on goal independence is defined, and a generalized version of Restricted AND-Parallelism (RAP) introduced as characteristic of this class. A simple and efficient backtracking algorithm for RAP is then discussed. An implementation scheme is presented for this algorithm which offers minimum overhead, while retaining the performance and storage economy of sequential implementations and taking advantage of goal independence to avoid unnecessary backtracking (*restricted intelligent backtracking*). Finally, the implementation of backtracking in sequential and AND-Parallel systems is explained through a number of examples.

KEYWORDS: LOGIC PROGRAMMING, PARALLEL PROCESSING, INTELLIGENT BACKTRACKING, AND-PARALLELISM, PROLOG.

1 Introduction

The execution of logic programs [9] in parallel is a subject of great interest because of the dual relationship between logic and parallelism: parallel execution seems to be a promising way of increasing the execution speed of logic programs; logic programs in turn offer multiple sources of parallelism [4] so that concurrency can (ideally) be uncovered automatically (or expressed cleanly) and mapped onto parallel architectures.

Of the several types of parallelism present in logic programs, we are specially interested in AND-Parallelism, because it offers the advantage that, in general, all work done by a collection of AND-Parallel processes is "useful" for finding a particular solution to a query. If OR-Parallelism is supported in addition to AND-Parallelism, backtracking is not needed; a set of "solutions" is maintained instead for each goal invocation. While the relative simplicity of this solution 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 it. Moreover, the additional parallelism often leads to combinatorial explosion of the search space.

1Department of Electrical and Computer Engineering, The University of Texas at Austin; Austin, TX 78712.

2Digital Equipment Corporation, Assigned Representative, Microelectronics and Computer Technology Corporation, Artificial Intelligence Program; 9430 Research Boulevard, Austin, TX 78759.

3This research was partially supported by the Microelectronics and Computer Technology Corporation (MCC).
As an alternative, we have presented a parallel abstract machine [7] [8] capable of implementing AND-Parallelism with very similar storage efficiencies and sequential-mode speed to that of the best sequential implementations. This is achieved in part by using backtracking rather than OR-Parallelism in the management of alternative paths in the search tree, and by implementing a stacking strategy with full space recovery on backtracking, as in sequential systems. In this paper we will deal with the problem of finding a suitable backtracking algorithm for this very general model of AND-Parallel execution, which can be implemented with minimum overhead, is compatible with the storage management strategy, and still takes advantage of the information available at run-time regarding goal independence in order to avoid unnecessary backtracking.

The organization of the paper is as follows: first, we will introduce "goal independence" models of AND-Parallelism and present a generalized version of Restricted AND-Parallelism (RAP) as a typical representative of this class. An efficient backtracking algorithm will then be elaborated for RAP. We will also study a suitable implementation strategy for this scheme capable of retaining most of the efficiency of sequential systems. Finally, some conclusions will be presented.

2 A General Model for AND-Parallelism: Goal Independence

Conery [5] showed 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. This can occur even in cases where the goals appear not to share variables at all. Consider the following clause:

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

During the resolution of a query of the form "?- crew(X,X) ." (looking for a person who is a pilot and can also operate a radio) X and Y in the clause above are coerced to be the same through unification. Thus, we cannot go ahead and evaluate "pilot(X)" and "radio_operator(Y)" in parallel (AND-Parallelism) because of the potential of conflicting instantiations for the identical variables X and Y.

Many approaches have been proposed in order to detect and deal with these variable binding conflicts either at compile-time or at run-time. Some of them, attempt to solve these conflicts without variable annotations and with minimal (or no) information from the user [5] [1] [6]. In other approaches, the user is required to annotate some variables or goals in the program in order to identify goals as "readers" or "writers" for each variable. This and other techniques are used in Concurrent Prolog [11], Parlog [2], IC-Prolog [3], Delta-Prolog [4] etc.

Although an interesting issue, we will not be concerned in this paper with the origin of these annotations. Instead, we will concentrate 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) and, in particular, on how backtracking can still be efficiently supported in such an environment4. Consequently, rather than analyzing a particular source-level language, 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 literal-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.

4This is in contrast with other approaches [11] [2] where *don't know* non-determinism has been given up in order to improve efficiency or simplify the implementation.