The &-Prolog System: Exploiting Independent And-Parallelism*

M. V. HERMENEGILDO**
Universidad Politécnica de Madrid (UPM),
Facultad de Informática,
28660-Boadilla del Monte, Madrid, Spain.
K. J. GREENE
MCC,
3500 W. Balcones Center Dr.,
Austin, Texas 78759, USA.

Received 4 December 1990
Revised manuscript received 6 August 1991

Abstract The &-Prolog system, a practical implementation of a parallel execution model for Prolog exploiting strict and non-strict independent and-parallelism, is described. Both automatic and manual parallelization of programs are supported. This description includes a summary of the system's language and architecture, some details of its execution model (based on the RAP-WAM model), and data on its performance on sequential workstations and shared memory multiprocessors, which is compared to that of current Prolog systems. The results to date show significant speed advantages over state-of-the-art sequential systems.

Keywords: Parallelism, Logic Programming, Prolog, Automatic Parallelization, Performance Analysis

§1 Introduction

There are many types of practical parallel processors currently available in the marketplace, and multiprocessor systems are expected to be the norm in the very near future. However, the amount of software that can exploit the performance potential of these machines is still very small. This is largely due to

** The work reported in this paper was performed in part at MCC's DC Lab. This author was supported in part by ESPRIT project 2471 “PEPMA” and CICYT project 361.208.
the difficulty in mapping the inherent parallelism in problems on to different multiprocessor organizations. There are in general at least two ways in which such a mapping can be performed: it can be done explicitly in the program by the user if the language includes parallel constructs, or it can be automatically uncovered by a "parallelizing" compiler from a program that has no explicit parallelism in it. Both approaches have their merits and drawbacks. A parallelizing compiler makes it possible to tap the performance potential of a multiprocessor without burdening the programmer. However, the capabilities of current parallelizing compilers are relatively limited, especially in the context of conventional programming languages. Parallelism explicitly expressed by the programmer using specialized constructs can be used when the programmer has a clear understanding of how the parallelism in the problem can be exploited. However, this adds in general an additional dimension of complication to the already complicated and bug-prone task of programming.* In reality, although experienced users may often have a correct intuition on which of the parts of a problem (and which of the parts of the associated program) can be solved in parallel, the task of correctly determining the dependencies among those parts and the sequencing and synchronization needed to correctly reflect such dependencies is proving to be very difficult and error-prone. This was recently also pointed out by Karp[19] who states that "the problem with manual parallelization is that much of the work needed is too hard for people to do. For instance, only compilers can be trusted to do the dependency analysis needed to parallelize programs on shared-memory systems."

Therefore, the best programming environment would appear to be one in which the programmer can freely choose between only concentrating on the conventional programming task itself (letting a parallelizing compiler uncover the parallelism in the resulting program) or, alternatively, performing also the task of explicitly annotating parts of the program for parallel execution. In the latter case, the compiler should be able to aid the user in the dependency analysis and related tasks. Ideally, different choices should be allowed for different parts of the program.

Karp also points out the lack of good parallelizing compilers, and predicts that the technology is still several years away. One reason for this is that the programming languages that are conventionally parallelized have a complex imperative semantics which makes compiler analysis difficult and forces users to employ control mechanisms that hide the parallelism in the problem. It is very

* In fact, the progress from systems which require from the programmer explicit creation and mapping of processes to a particular processor interconnection topology and extensive granularity control, to systems which don't require at least some of these tasks appears to be a leap forward comparable to the appearance of the concept of virtual memory (compared to overlays) or even high-level languages (compared to programming in machine code). Of course, in the same way as there is sometimes a case for assembler programming in parts of a program which are particularly performance sensitive, there will be some cases in which complete explicit control of parallelism is indicated.