Automatic Unrestricted Independent And-Parallelism in Declarative Multiparadigm Languages

by

Amadeo Casas

B.S., Computer Science, University of Valladolid, 2003
M.S., Computer Engineering, University of New Mexico, 2005
M.B.A., Robert O. Anderson School of Management, 2008

Advisors: **Manuel V. Hermenegildo**
Manuel Carro

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Engineering

The University of New Mexico

Albuquerque, New Mexico

December, 2008

To my family, por su apoyo infinito y comprensión.

Acknowledgments

The making of this thesis would not have been possible without the help and support of many people and I would like to take this opportunity to express my gratitude to all of them.

First, I would like to thank my advisor, Manuel V. Hermenegildo, for introducing me to the truly rewarding world of research and for his stimulating and invaluable help with this thesis. He has my sincere admiration for his knowledge in the field, which has been and is a great inspiration to me. I am deeply grateful for his patience with me and all the encouragement I received during the process of the becoming of this thesis. Secondly, I want to thank Manuel Carro, for taking effort in providing me with valuable comments and suggestions during the writing of my thesis. His untiring help and guidance throughout my research has had great importance for the final outcome.

I would also like to thank Mario Méndez-Lojo and Jorge Navas for those interesting scientific discussions and, additionally, for our absorbing conversations on all serious things in life. Furthermore, I have greatly appreciated them for being a source of good humour in moments of frustration.

Furthermore, I am thankful to Iberdrola for funding my research through the Prince of Asturias Chair in Information Science and Technology at UNM.

I also wish to show gratitude to the professors in the Electrical and Computer Engineering Department at the University of New Mexico for teaching me so many important things, which I have brought into use during my research. In particular, I would like to warmly thank Gregory L. Heileman and Thomas P. Caudell for being part of the committee and showing interest in my work. Furthermore, I owe thanks to Deepak Kapur and his research group for their motivating and inspiring Logic Group meetings and classes. In this context, I would like to thank my officemate Stephan Falke for creating a friendly work atmosphere.

Moreover, I am grateful to all members of the CLIP research group from Madrid who provided me with already existing tools which proved to be highly useful in my research. In particular, I would like to thank Daniel Cabeza for his implementation and work in the functional syntax of Ciao.

Finally, the completion of this thesis would not have been possible without the support of my family and friends. I will be eternally thankful to my parents for the great values of responsibility and hard work that they have taught me throughout my life. Thank you to Sine, for her love, inexhaustible patience with my grumpy moods and her success in cheering me on. Also, a very heartfelt thank you to all my friends back home for still being there when needed and to all my friends in Albuquerque for being able to put a warm smile on my face!

As a coda, there are people who have influenced my work which I cannot herein mention. I apologize and guarantee that your help has not gone by unappreciated.

Amadeo Casas
September 2008

Automatic Unrestricted Independent And-Parallelism in Declarative Multiparadigm Languages

by

Amadeo Casas

ABSTRACT OF DISSERTATION

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

Engineering

The University of New Mexico

Albuquerque, New Mexico

December, 2008

Automatic Unrestricted Independent And-Parallelism in Declarative Multiparadigm Languages

by

Amadeo Casas

B.S., Computer Science, University of Valladolid, 2003

M.S., Computer Engineering, University of New Mexico, 2005

M.B.A., Robert O. Anderson School of Management, 2008

Ph.D., Engineering, University of New Mexico, 2008

Abstract

Parallelism capabilities are becoming ubiquitous thanks to the widespread use of multi-core processors. This has renewed the interest in language-related designs and tools which can simplify the task of producing parallel programs. The use of declarative languages is considered to be an interesting and promising approach for increasing performance through the execution of parallel programs. In particular, nondeterminism and partially instantiated data structures give logic programming expressive power beyond that of functional programming. However, functional programming often provides convenient syntactic features, such as having a designated

implicit output argument, which allow function call nesting and sometimes results in more compact code, as well as sometimes a more direct encoding of lazy evaluation, with its ability to deal with infinite data structures. The high-level nature of these languages, in addition to their relatively simple semantics and the use of logic variables, preserves more of the original parallelism to be uncovered by an automatic parallelization. Different alternatives for performing automatic goal-level, unrestricted independent and-parallelization of logic programs through source-to-source transformations are studied in this work, which uses as targets new parallel execution primitives which are simpler and more flexible than the well-known fork-join parallel operator, in order to generate better parallel expressions by exposing more potential parallelism among the literals of a particular program. An alternative approach for implementing and-parallel logic programming languages has also been explored, which tames the complexity of the low-level machinery required by most of the previous implementations by raising core parts to the source language level. A significant portion of the implementation mechanisms of parallel execution is handled directly at the Prolog level with the help of a comparatively small number of concurrency-related primitives that take care of simpler low-level tasks such as locking, and thread and stack set management. Moreover, in order to extend this work to different paradigms, a syntactic functional extension to ISO-standard Prolog systems has been developed, which covers function application, predefined evaluable functors, functional definitions, quoting, and lazy evaluation, and is composable with higher-order and other extensions to ISO-Prolog, such as constraints.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction

This chapter introduces a general overview and the main motivations that encouraged me to perform the work presented in this thesis. In addition, the objectives and the actual contributions made during its development are listed. Finally, the organization of the thesis will be outlined.

1.1 Overview and Motivation

New multicore technology is challenging developers to create applications that take full advantage of the power provided by these processors. The path of single-core microprocessors following Moore's Law has reached a point where very high levels of power and, as a result, heat dissipation are required to raise clock speeds. Multicore systems seem to be the main architectural solution path taken by manufacturers for offering potential increases in performance without running into these problems. Nowadays, most laptops on the market contain two cores, capable of running up to four threads simultaneously, and single-chip, 8-core servers are now in widespread

use. Furthermore, the trend is that the number of on-chip cores will double with each processor generation.

This wide availability of multicore processors is finally making parallelism mainstream. In this context, being able to exploit such parallel execution capabilities in programs as easily as possible becomes more and more of a necessity, since applications that are not parallelized will show little or no improvement in performance as new generations with more processors are developed. In fact, it is well-known [KB88] that parallelizing programs is a hard challenge. One of the main issues that make the widespread use of parallelism difficult is that few applications are written to exploit parallelism. Thus, there is renewed research interest in the development and design of languages and tools to simplify the task of writing parallel programs. This includes the design of languages that provide a better support for the exploitation of parallelism, libraries that offer support for parallel execution, and *parallelizing compilers* capable of helping in the parallelization process.

The focus of this thesis is on the issue of automatic parallelization. Parallelizing compilers can dramatically reduce the burden on the programmer when parallelizing programs. However, although there is hope that parallelizing compilers may eliminate the need of manual parallelization, significant challenges still remain to be solved in this area, as, for instance, dealing appropriately with irregular computations, handling speculation, dealing with complex data structures and pointers, and developing new parallelization techniques for higher-level programming languages.

Due to the particular features of the programming paradigms, the amount of progress made within the topics related to automatic parallelization differs. However, despite the inherent differences between imperative, object-oriented and declarative languages, the issues to be tackled are quite similar, and thus the results and solutions obtained in the study of automatic parallelization of a particular program-

ming paradigm can speed up the development process of more efficient parallelizing compilers for all programming paradigms.

In particular, declarative languages have been traditionally considered an interesting approach for obtaining increased performance through parallel execution on multicore architectures, including multicore embedded systems. Among them, logic programming (LP) offers a number of features, such as nondeterminism and partially instantiated data structures, which give it expressive power beyond that of functional programming. However, certain aspects of functional programming (FP) provide in turn syntactic convenience. This includes, for instance, having a syntactically designated output argument, which allows the usual form of function call nesting and sometimes results in more compact code. Also, lazy evaluation, which brings the ability to deal with infinite non-recursive data structures [Nar90, Ant91], while subsumed operationally by logic programming features such as delay declarations, enjoys a more direct encoding in functional programming [Her00].

This thesis presents a different alternative for exploiting parallelism in logic programs, which is based on breaking down the concept of the traditional fork-join parallelization into simpler alternatives that are able to provide more flexibility in the parallel annotation of the original code, which in addition results in better performance results. Also, this thesis proposes a high-level implementation of the execution model for independent and-parallel programs, which relies on a minimal set of concurrency-related primitives and provides flexible solutions for some of the main problems found in previous and-parallel implementations, while avoiding most of the low-level machinery required by other previous proposals. Finally, in order to study and expand the obtained performance results to some other paradigms, and be able to support parallel execution in both logic and functional languages, a syntactic functional extension to the system has been developed. Bringing this syn-

tactic convenience to logic programming actually results in a more compact program representation.

1.2 Thesis

My thesis is that flexible, extensible and maintainable high-performance high-level solutions can be developed for unrestricted independent and-parallelism in declarative multiparadigm languages. In order to support my thesis I have designed, developed, implemented and evaluated solutions for automatic parallelization in these languages based on a logic programming kernel, and aimed at recovering a significant part of the efficiency lost with the high-level execution model.

1.3 Goals and Contributions

This section presents the main objectives of this thesis and the contributions made during its development. In addition, the collaborations with other researchers, and the publications resulting from them, are mentioned.

The objectives and contributions of this thesis can be structured in three main topics, as follows:

Functional Syntactic Layer for Logic Programming: In order to support both logic and functional programming, this thesis presents a design for an extensive functional syntactic layer for logic programing, with Prolog systems in mind, as well as its implementation in the Ciao system [BCC+06]. While the idea of adding functional features to logic programming systems is clearly not new, and there are currently a good number of systems [NM88, CKW93, Han,

MNRA89, SHC96, BdlBM+06, HF00] which integrate functions and higher-order programming into some form of logic programming, the proposal of this thesis offers a combination of features which make it interesting in itself. The approach is completely syntactic, functions can be limited or retain the power of predicates, any predicate can be called through functional syntax, lazy evaluation is supported both for functions and predicates, functional syntax can be combined with other extensions and thus inherit all other possible syntactic and semantic extensions, such as higher-order, assertions, records or constraints. Also, because of the syntactic nature of the extensions they can be the target of analysis, optimization, static debugging, verification, etc., as performed by, e.g., the Ciao preprocessor [HPBLG05], without any modification to the compiler or abstract machine.

This approach was motivated by some of the language extension capabilities of the Ciao system [BCC+06]: Ciao offers a complete ISO-Prolog system, but one of its most remarkable features is that, through a novel modular design [CH00], all ISO-Prolog features are library-based extensions to a simple declarative kernel. This allows on one hand the possibility to avoid loading any (for instance, impure) features from ISO-Prolog when not needed, and on the other hand adding many additional features at the source (Prolog) level, without modifying the compiler or the low-level machinery. The facilities that allow this, grouped under the Ciao *packages* concept [CH00], are the same ones used for implementing the functional extensions proposed herein, and are also the mechanism by which other syntactic and semantic extensions are supported in the system. The latter includes constraints, objects, feature terms/records, persistence, several control rules, etc., resulting in Ciao being a publicly licensed, next generation multi-paradigm programming environment.

However, while the Ciao extension mechanisms make implementation smoother

and more orthogonal, a fundamental design objective and feature of the functional extensions is that they are to a very large extent directly applicable to, and also relatively straightforward to implement in, any modern ISO-Prolog system [DEDC96]. Therefore, the corresponding contributions made in this thesis can be adopted in such systems.

This work has been done in collaboration with Prof. Daniel Cabeza (Technical University of Madrid), who implemented the functional syntax package in Ciao. This work has been published in the following international conference and workshop:

- [CCH06], written together with D. Cabeza and M. Hermenegildo. Presented at the *Eighth International Symposium on Functional and Logic Programming (FLOPS'06)*. April 2006.
- [CCH05], written together with D. Cabeza and M. Hermenegildo. Presented at the *Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS'05, ICLP associated workshop)*. October 2005.

Annotators for Unrestricted Independent And-parallelism: One particularly successful approach to automatically parallelizing a logic program uses three different stages. The first one detects the dependencies between pairs of procedure calls in the original program. The second stage performs global analysis [BdlBH99] over the program in order to gather static information on its procedure calls. Finally, the third stage transforms the original program into a parallel version by *annotating* it with parallel execution operators. This annotation should respect the dependencies found in the original program while, at the same time, exploit as much parallelism as possible.

This annotation process is the focus of the contribution of this thesis in the

area of automatic parallelization. Transformation algorithms used in previous approaches in the context of logic programming have used nested fork-join style parallelization, which has the drawback that they sometimes have to give up parallelizing some goals due to the somewhat rigid structure imposed on the final program. This limitation guided the development of new annotation algorithms which target and-parallelism primitives that can express richer dependency graphs than those which can be encoded with the *nested fork-join* approaches.

Limitations of the fork-join annotations have been previously studied for imperative languages [Sar90], in which reordering of instructions was studied to expose the maximum amount of parallelism, however the non-deterministic nature of logic programs was not covered. The work presented in this thesis shows that, since the transformed programs will contain in some cases more parallelism, it will be possible to obtain better speedups than the *fork-join* variants for such cases.

This work has resulted in an accepted publication in the following international conference:

- [CCH07a], written together with M. Carro and M. Hermenegildo. Presented at the *17th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'07)*. August 2007.

An extended version of this paper is currently submitted for publication in an international journal.

High-Level Model for Unrestricted Independent And-parallelism: Most of the previously developed run-time parallel systems, aimed at exploiting and-parallelism [Her86, HG91, BLOO86, Kal87b, She96, GHPSC94, Jan94, SCWY91,

San93] are based on an extension of the original WAM architecture and set of instructions, and were originally implemented, as most of the other systems mentioned, on shared-memory multiprocessors. While these models and their implementations have been shown very effective at exploiting parallelism efficiently and obtaining significant speedups [HG91, PG98], most of them are based on quite complex, low-level machinery which included an extension of the WAM instructions, and new data structures and stack frames in the stack set of each agent, which makes the implementation and maintenance, and also the extensibility of these systems inherently hard.

This fact motivated the design of an alternative approach which is aimed at taming that complexity by raising core parts of the implementation to the source language level, and relying only on a comparatively small number of concurrency-related primitives which take care of lower-level tasks. This thesis will present an implementation model for independent and-parallelism which fully supports non-determinism through backtracking and provides flexible solutions for some of the main problems found in previous and-parallel implementations. In addition, this solution is able to optimize the execution for the case of non-failing deterministic programs and to exploit *unrestricted* and-parallelism, which allows exposing more parallelism among clause literals than fork-join-based proposals.

A performance hit could be of course expected due to the high-level nature of the implementation. However, several performance results presented in this thesis will show that this approach does not necessarily incur insurmountable efficiency loses. Moreover, that division of concerns will make it possible to more easily explore variations on the execution schemes.

The contributions made within this work have been accepted for publication in the following international conferences and workshops:

- [CCH08a], written jointly with M. Carro and M. Hermenegildo. To be presented at the *24th International Conference on Logic Programming (ICLP'08)*. December 2008.
- [CCH08b], written jointly with M. Carro and M. Hermenegildo. Presented at the *10th International Symposium on Practical Aspects of Declarative Languages (PADL'08)*. January 2008.
- [CCH07c], written jointly with M. Carro and M. Hermenegildo. Presented at the *Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS'07, ICLP associated workshop)*. September 2007.
- [CCH07b], written jointly with M. Carro and M. Hermenegildo. Presented at *Parallel Symbolic Computation (PASCO'07)*. July 2007.

1.4 Organization

This thesis is organized as follows. Chapter 1 has introduced the main motivations of this work and a summary of the main goals and contributions. Chapter 2 presents the main technical background which is necessary for a good understanding of the rest of the chapters in this thesis. Chapter 3 explains the design of the syntactic approach to the functional notation and lazy evaluation, and provides several examples of how to combine them with other syntactic and semantic extensions of the language. Chapter 4 presents several annotation algorithms which are able to exploit more of the intrinsic parallelism in a program, proves their correctness and shows their performance with a comparison with some of the previous annotation algorithms used. Chapter 5 describes the high-level implementation of the execution model for unrestricted independent and-parallelism. Finally, Chapter 6 summarizes the

conclusions obtained with this work and highlights some of the possible directions for future research.

Chapter 2

Background

This chapter presents the main technical background necessary to understand the following chapters of this thesis.

2.1 Parallelization in LP

Declarative languages, and among them logic programming languages, and also the new multiparadigm languages based on logic programming kernel languages, offer a particularly interesting case study for the area of automatic parallelization [GPA+01].

The comparatively higher level of abstraction of declarative languages allows writing programs which are closer to the specification of the solution. Besides, there is often more freedom in the implementation of different operational semantics which respect the declarative semantics. In particular, the notion of control in declarative languages frequently is separated from the actual specification, which allows for more flexibility to arrange the evaluation order of some operations, including executing them in parallel if convenient, without affecting the semantics of the original program.

Additionally, the cleaner semantics that declarative programs enjoy and the use of logic variables, which can be assigned only one value, makes it possible to automatically detect more accurately any lack of dependencies among operations and hence to exploit opportunities for parallelism more easily than in other programming paradigms, in comparison with imperative languages. For instance, it is not necessary to check for some types of flow dependencies or to perform single statement assignment (SSA) transformations. At the same time, the presence of dynamic data structures with "declarative pointers" (logical variables), irregular computations, and complex control makes the parallelization of logic programs a particularly interesting case, since it allows tackling the more complex challenges faced in the parallelization of other programming paradigms (including imperative programming) but in a somewhat simpler and semantically well-understood context [Her00].

Finally, the parallelization of logic programs also brings unique challenges related to backtracking and the fact that multiple solutions may be produced for each procedure call.

2.2 Types of Parallelism in LP

Because of this potential, quite significant progress has been made in the area of parallelization of logic programs [GPA$^+$01], where two main forms of parallelism have been studied, presented in the following sections.

2.2.1 Or-Parallelism

Or-parallelism [LH85, War87b, Lus88, Kal87a, Sze89, War87a, AK90a, CH83, Hau90] is exploited when the alternatives created by non-deterministic goals are explored

simultaneously by different processors, in order to reduce the time taken to traverse their (possibly large) search space. The exploitation of this type of parallelism is interesting in applications that involve extensive seach, since the choices that are represented by alternative clauses usually involve a large number of steps before a failure or a success in the search occurs. Some of the most relevant or-parallelism systems are Aurora [Lus90] and MUSE [AK90b].

Example 1 *The following program solves the* 8-queens *problem:*

```
queens(Queens) :-
        solve_queens([8, 7, 6, 5, 4, 3, 2, 1], [], Queens).

solve_queens([], Qs, Qs).
solve_queens(Unplaced, Placed, Qs) :-
        select_queen(Q, Unplaced, NewUnplaced),
        no_attack(Q, Placed),
        solve_queens(NewUnplaced, [Q|Placed], Qs).

select_queen(X, [X|Ys], Ys).
select_queen(X, [Y|Ys], [Y|Zs]) :-
        select_queen(X, Ys, Zs).

no_attack(Q, Safe) :- no_attack_acc(Safe, Q, 1).
no_attack_acc([], _, _).
no_attack_acc([Y|Ys], Queen, Nb) :-
        Queen =\= Y + Nb,
        Queen =\= Y - Nb,
        Nb1 is Nb + 1,
        no_attack_acc(Ys, Queen, Nb1).
```

The predicate `select_queens/3` *will create alternatives to the execution of the algorithm in each of the recursive calls to the predicate* `solve_queens/3`*. In or-parallelism, these alternatives may be executed in parallel by different agents.*

This type of parallelism in logic programming is practically solved, since it is conceptually simpler than and-parallelism.

2.2.2 And-Parallelism

An alternative strategy that is used to parallelize logic programs is referred to as *and-parallelism* [DeG84, Kal87a, BSY88, CDD85, Her86, Lin88, WR87, WW88, BR86, Con83, Fag87, Hua85, LK88, PK88], which aims at executing simultaneously conjunctive goals in clauses or in the resolvent, in a similar fashion as traditional parallelism. While or-parallelism can only obtain speedups when there is search involved, and-parallelism can be used in more algorithmic schemes, with loop parallelization, and divide-and-conquer and map-style algorithms being classic representatives. Examples of systems that have exploited and-parallelism are &-Prolog [Her86, HG91], ROPM [Kal87b], AO-WAM [BLOO86] and DDAS [She96]. Additionally, some systems such as ACE [GHPSC94], AKL [Jan94], and Andorra-I [SCWY91, San93] exploit certain combinations of both and- and or-parallelism.

Example 2 *The following program presents an and-parallel version of the* quicksort *algorithm:*

```
qsort([], []).
qsort([X|L], R) :-
        partition(L, X, L1, L2),
        qsort(L2, R2) &
```

```
        qsort(L1, R1),
        append(R1, [X|R2], R).

partition([], _B, [], []) :- !.
partition([E|R], C, [E|Left1], Right) :-
        E < C,
        !,
        partition(R, C, Left1, Right).
partition([E|R], C, Left, [E|Right1]) :-
        E >= C,
        partition(R, C, Left, Right1).
```

The only difference with the sequential version of this algorithm is the substitution of the sequential operator (`,/2`) by the `&/2` operator in order to schedule both recursive calls to `qsort/2` to be executed in parallel.

The `&/2` operator in the example above represents the traditional fork-join nested parallelism. This type of parallelization is referred to as *restricted and-parallelism (RAP)*. This operator has been adopted from the &-Prolog model [HG91], in which conjunctions which are to be executed in parallel are often marked by replacing the sequential comma (`,/2`) with a parallelism operator (`&/2`).

Since and-parallelism corresponds to the classical parallelism found in other programming paradigms, the work presented in this thesis will concentrate on the issue of automatic and-parallelization. From now on in this thesis, the term parallelism will be referring to and-parallelism.

2.3 Classical Approaches to And-Parallelism

The main objective of a parallelizing compiler is to uncover as much parallelism as possible, but typically preserving some conditions to guarantee that the set of solutions obtained is the same one as in the sequential execution and that there is not a decrease in the performance of the execution, i.e., that the parallel execution is never slower than the sequential execution. Thus, a correct parallelization has been traditionally defined as one that preserves during and-parallel execution some key properties, which are typically: [HR95]

Correctness: the set of solutions returned by the parallel execution of a particular program is the same as its sequential execution.

No-slowdown: the execution time of the parallel execution is less than, or equal to, the execution time of the sequential execution, assuming there is not overhead in the parallel execution.

Not only errors but also significant inefficiency can arise from the simultaneous execution of computations which depend on each other since, for example, this may trigger more backtracking than in the sequential case. Thus, the preservation of these properties is ensured by executing in parallel goals which meet some non-unique notion of *independence*, meaning that the goals to be executed in parallel do not interfere with each other in some particular sense. This can include, for instance, absence of competition for binding variables among goals to be run in parallel plus other considerations such as, e.g., absence of side effects. For simplicity, it is assumed that the programs to be parallelized are free of side-effects. Note however that this does not affect the generality of the presentation, as dependencies are analyzed in a generic way.

Therefore, goals are said to be *independent* if their parallel execution will not perform additional search and will not produce incorrect results, i.e., if they preserve the *correctness* and *no-slowdown* properties. Very general notions of independence have been developed, based on constraint theory [dlBHM00]. However, for simplicity, only those based on variable sharing will be discussed.

There are two main models in and-parallelism:

Dependent and-parallelism (DAP): goals are executed in parallel even if they share variables, and the competition to bind them has to be dynamically dealt with using notions such as sequencing bindings from producers to consumers. It is a very interesting model, but unfortunately it usually implies substantial execution overhead.

Independent and-parallelism(IAP): One of the best understood sufficient conditions for ensuring that goals meet the efficiency and correctness criteria for parallelization is called *strict independent and-parallelism (SIAP)* [DeG84, HR89], which entails the absence of shared variables at runtime between any two literals being parallelized. It should be noted that some proposals exploit and-parallelism between goals which do not meet this condition, but on which other restrictions are imposed which also ensure the no-slowdown property and absence of conflicts due to the binding of shared variables. An example of such restrictions is *non-strict independent and-parallelism (NSIAP)* [HR90, HR95], in which two goals share some variables, although there is no competition in their bindings. Although non-strict independence between two literals cannot be determined by inspecting the previous state of execution, and thus global analysis of the original program is required, it is quite interesting because it uncovers some of the parallelism that is present in applications that manipulate open data structures, as for instance difference lists. Another example

is determinacy-based independence, as used for example in the Basic Andorra Model [SCWY91], which makes use of determinism information in order to decide whether two goals are to be executed in parallel or not, since two computations that have no alternatives to execute, and their execution is known to never fail, are independent and can thus be executed in parallel. In addition, an interesting issue is at what level of granularity the notion of independence is applied: at the goal level, at the binding level, etc. These concepts of independence have been generalized through the notion of search space preservation [dlB94] and local independence [BHMR94, BHMR98], and also extended to constraint logic programming [dlBHM93, dlBHM00], and constraint logic programming with dynamic scheduling [dlBHM96].

Example 3 *The following code presents a parallel solution for the* hanoi *problem:*

```
hanoi(1, A, _, C, [mv(A,C)]).
hanoi(N, A, B, C, M) :-
        N > 1,
        N1 is N - 1,
        hanoi(N1, A, C, B, M1),
        hanoi(N1, B, A, C, M2) &
        append(M1, [mv(A,C)], T),
        append(T, M2, M).
```

In this case, the second recursive call to ***hanoi/5*** *will be executed in parallel with the first call to* ***append/3****. That will produce a correct parallelization since both calls are independent, i.e., they do not share variables and thus conflicts are avoided. Also, note that some other parallelizations are also possible in this algorithm.*

In particular, both recursive calls to ***hanoi/5*** *could be executed in parallel, but that*

would require removing the parallelization with the first call to `append/3`*, due to the dependency created via the variable* `M1`*.*

This thesis will focus on both strict and non-strict independent and-parallelism, as both have practically identical implementation requirements.

Also, the work performed in this thesis has followed and extended some of the ideas and the architecture of the &-Prolog model. It basically consisted of two components: a parallelizing compiler which detects the possible runtime dependencies between goals in clause bodies and annotates the clauses with expressions to decide whether parallel execution can be allowed at runtime, and a run-time system that exploits that parallelism. The following sections will provide a general overview of these two components, since they will be the basis for this work.

2.4 CDG-Based Automatic Parallelization

Figure 2.1 presents an overview of the process that has been followed in the work presented in this thesis to automatically transform a Prolog program into a semantically equivalent parallel version of it [HW87, BdlBH99, GPA+01]. It uses three different stages. The first stage explores the literals in the original clause searching for candidates for parallel execution by detecting data and control dependencies between pairs of those literals. A directed dependency graph (see Figure 4.1(b) as an example) is then built to capture this extracted information. The nodes in the graph correspond to literals in the body of the clause and the edges represent dependence conditions between them. Edges are labeled with the associated dependence conditions (which may be trivially *true* or *false*).

As a second stage, a global analysis [JL89, MH89, MH92, JL92, BdlBH99] can

be run in order to gather information regarding, e.g., variable aliasing, groundness, and side effects, in order to prove statically whether those dependence conditions are statically true or false. Those edges whose dependence condition becomes *true* are eliminated from the graph, since the two literals are independent. If an edge condition becomes *false* then it will be left in the graph as an unconditional edge, since the two literals are dependent. For the rest of the edges in the graph, when a condition cannot be completely evaluated at compile-time, it may remain associated to the edge, but possibly in a simplified form.

Finally, the third stage corresponds to the annotation process, which encodes the resulting dependency graph in the target parallel language. This annotation should respect the dependencies implied by the graph while, at the same time, exploiting as much parallelism as possible. Several algorithms based on different heuristics have been proposed to compile the dependency graph into parallel code [DeG87, MBdlBH99, MH90, BdlBH94, CH94] using fork-join structures. In this process, labeled edges result in run-time checks when conditional parallel expressions are allowed. Since the tasks to be parallelized may not represent a sufficient amount of computation with respect to the overhead that is incurred when the run-time check is evaluated, unresolved dependencies are sometimes assumed to always hold, and parallel execution will be allowed only between literals which have been statically determined to be independent. This approach brings simplicity and avoids potentially costly run-time checks in the parallelized code at the expense of potentially losing some parallelism.

Two main forms of annotating parallel code are presented in the following sections: *restricted IAP* and *unrestricted IAP*.

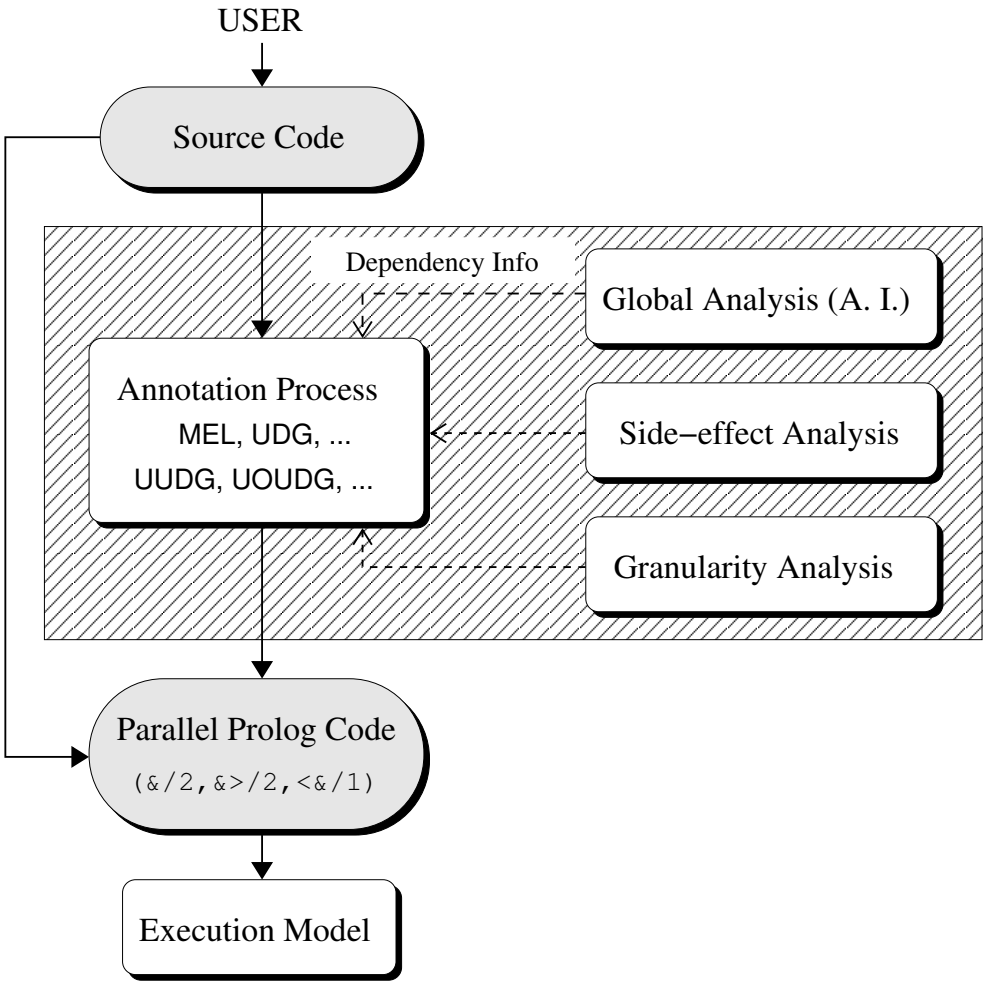


Figure 2.1: Simplified scheme of the parallelizing compiler architecture.

2.4.1 Restricted IAP

The &-Prolog language [Her86, HG91] has been a frequent vehicle for expressing goal-level, restricted independent and-parallelism in logic programs. It is basically an extension of Prolog, adding parallel expressions (*ParExp* in the grammar below). In its *restricted* version, an associative parallel fork-join operator `&/2` is used. Essentially, the sequential comma can be replaced with this operator in order to mark goals which can be executed in parallel. Programs parallelized in the &-Prolog language

typically also use the standard Prolog if-then-else constructions to be able perform computations in different ways (e.g., sequentially or in parallel) depending on the result of evaluating some independence-related run-time checks (such as independence or groundness), or granularity control-related tests (such as data size checks). A simplified grammar (i.e., without cuts, side-effects, and other built-ins) defining the syntax of restricted &-Prolog programs follows:

Definition 1 (Restricted &-Prolog grammar) *Let $\overrightarrow{t}$ be a tuple of terms and p a predicte symbol. Then, the following grammar defines the set of valid sentences in the restricted &-Prolog language:*

Program ::= Clause `.` Program | ε
Clause ::= Literal | Literal `:-` Body
Body ::= Literal | Literal `,` Body | Body `->` Body `;` Body | ParExp
ParExp ::= Body `&` Body
Literal ::= $p(\overrightarrow{t})$

Programs parallelized in the &-Prolog language typically also use standard Prolog if-then-else constructions (conds `->` body1 `;` body2), to be able perform computations in different ways (e.g., sequentially or in parallel) depending on the result of evaluating some independence-related run-time checks (such as independence or groundness), or granularity control-related tests (such as data size checks).

In addition, the operational semantics of the fork-join operator `&/2` are as follows [HR89, MBdlBH99]:

Definition 2 (Operational semantics of `&/2`) *Let $\langle g, s\rangle$ be the computational state, where g is a goal and s is a store. The operational semantics of the state of com-*

putation $\langle (g_1 \texttt{ \& } \ldots \texttt{ \& } g_n).c,\ s\rangle$*, where* c *is the continuation of the parallel computation, is given by the parallel computation of the states* $\langle g_1,\ s_1\rangle, \ldots, \langle g_n,\ s_n\rangle$*, if* $\langle \varepsilon,\ s \wedge s_1\rangle, \ldots, \langle \varepsilon,\ s \wedge s_n\rangle$ *and* $\langle c,\ s \wedge s_1 \wedge \ldots \wedge s_n\rangle$*.*

However, the restricted &-Prolog language has some limitations in the parallelization of some particular CDGs, which relies on the use of the fork-join operator, which is a rigid operator in the sense that it does not allow to execute some other literals until the execution of both parallel goals has finished, and furthermore this can prevent exploiting some of the parallelism that is implicitly represented in the dependency graph.

Section 4.1 will further explain this limitation and provide some examples in order to acquire a better understanding of it.

2.4.2 Unrestricted IAP

As previously explained, some and-parallel systems rely on the use of the fork-join operator (`&/2`) as the most basic construction to exploit parallelism between goals which are independent at run-time, because of the simplicity in which parallel computations can then be described. This section presents an extension of the restricted &-Prolog language introduced in Section 2.4.1:

Definition 3 (Unrestricted &-Prolog grammar) *Let* $\overrightarrow{t}$ *be a tuple of terms and* p *a predicte symbol. Then the following grammar defines the set of valid sentences in the unrestricted &-Prolog language:*

Program ::= Clause `.` Program | ε

Clause ::= Literal | Literal `:-` Body

Body ::= Literal | Literal `,` Body | Body `->` Body `;` Body | ParExp

ParExp ::= Body `&` Body | Body `&>` Handler | Handler `<&`

Handler ::= Literal

Literal ::= $p(\overrightarrow{t})$

The changes with respect to the previous language are the addition of two new expressions for the production corresponding to the non-terminal *ParExp* and a production for the non-terminal *Handler*. This new language will be referred to as the *unrestricted* &-Prolog language. The new productions extend the parallel expressions used in the restricted &-Prolog language by adding two more basic constructions [CH96, Cab04] to schedule goals for parallel execution, `&>/2` and `<&/1`, informally defined as follows:

Definition 4 (Publish operator) `Goal &> H` *schedules goal* `Goal` *for parallel execution and continues executing the code after* `Goal &> H`. `H` *is a* handler *which contains (or* points to*) the state of goal* `Goal`*.*

Definition 5 (Wait operator) `H <&` *waits for the goal associated with* `H` *to finish, or executes it if it has not been taken by another thread yet. After that point any bindings made for the output variables of the goal associated to* `H` *are available to the executing thread.*

Furthermore, the following definitions will present the operational semantics of these two operators:

Definition 6 (Operational semantics of `&>`/2) *Let $\langle g, s\rangle$ be the computational state, where g is a goal and s is a store. The operational semantics of the state of computation $\langle (g_1$ `&>` $h).c,\ s\rangle$, where c is the continuation of `&>`/2, is given by the computational state $\langle c,\ s\rangle$.*

Definition 7 (Operational semantics of `<&`/1) *Let $\langle g, s\rangle$ be the computational state, where g is a goal and s is a store. The operational semantics of the state of computation $\langle (h_1$ `<&`$).c,\ s\rangle$, where h_1 is the handler associated to goal g_1 and c is the continuation of the parallel computation, is given by the parallel computation of the state $\langle g_1,\ s_1\rangle$, if $\langle \varepsilon,\ s \wedge s_1\rangle$ and $\langle c,\ s \wedge s_1\rangle$.*

`G &> H` ideally takes a negligible amount of time to execute, although the precise moment in which `G` actually starts depends on the availability of resources (primarily, free agents/processors). On the other hand, `H <&` suspends until the associated goal finitely fails or returns an answer. It is interesting to note that the approach shares some similarities with the concept of *futures* in parallel functional languages. A future is meant to hold the return value of a function so that a consumer can wait for its complete evaluation. However, the notions of "return value" and "complete evaluation" do not make sense when logic variables are present. Instead, `H <&` waits for the moment when the producer goal has completed execution, and the "received values" (a tuple, really) will be whatever (possibly partial) instantiations have been produced by such goal.

In addition, it is necessary to highlight that actual backtracking is performed at `H <&`, and the memory reserved by the handler needs to be released when `G &> H` is reached on backtracking. If `G &> H` is reached on backtracking but `H <&` was not reached on forward execution, this means that some of the goals between these two points has failed without a solution, and the execution of goal `G`, whatever its state is, must be cancelled, since its result will not be needed anymore.

Also, although exception handling is beyond the scope of this thesis, in general exceptions uncaught by a parallel goal surface at the corresponding `<&/1`, where they can be captured by the parent.

With the previous definitions, the restricted fork-join operator `&/2` can conceptually be written in terms of the unrestricted operators `&>/2` and `<&/1` as:

```
A & B :- A &> H, call(B), H <&.
```
(2.1)

This is an important result, since it indicates that any parallelization performed using the operator `&/2` can be made using the operators `&>/2` and `<&/1` without loss of parallelism.

Note that this does not preclude any implementation from trying to make `&/2` as efficient as possible —for example, by expanding `&/2` inline following the above definition, or by giving lower-level explicit mechanisms to implement `&/2`. In any case, the above definition clearly indicates that any parallelization performed using `&/2` can be made using `&>/2` and `<&/1` without loss of parallelism.

Also, literal `B` is executed locally because when running the very common tail-recursive case `p :- q & p`, one wants the recursive call `p`, the 'generator' of parallel goals, to be executed with no delays in order to spawn parallel `q`'s as quickly as possible. This reverses the order of solutions with respect to the sequential case, but this is not a real problem for pure goals.

Chapter 4, and more in particular Section 4.1, will go further with the motivations for using these unrestricted operators, and in addition will provide some examples to show their effectiveness in the cases where the restricted fork-join operator misses some of the possible parallelization in the program.

2.5 Implementation Tools: The Marker Model

The &-Prolog model [HG91] was the first complete description of a parallel Prolog system capable of achieving effective speedups with respect to the best sequential systems, and its basic ideas have been adopted by many previously mentioned systems as, for instance, &ACE [GHPSC94] and DDAS [She96]. This section will now focus on providing a general outline of the actual execution model of parallel logic programs.

The execution model introduced by &-Prolog [HG91] is referred to as the *multi-sequential, marker model*. This implementation approach has been adopted by many and-parallel systems, for both IAP [HG91, PGH95] and DAP [She96], to execute goals in parallel. In this model, parallel goals are executed in different *abstract machines* which run in parallel. In order to preserve the sequential speed, these abstract machines are extensions of the sequential model, usually the *Warren Abstract Machine (WAM)* [War83, AK91], which is the basis of most efficient sequential implementations.

Herein it is assumed for simplicity that each WAM has a parallel thread (an "agent") attached and that there are as many threads as processors. Thus, WAMs, agents, or processors can be referred to interchangeably. In this model, within each WAM, sequential fragments appear in contiguous stack sections exactly as in the sequential execution. However, in some proposals this need not be so: *continuation markers* [SH96] allow sequential execution to spread over non-contiguous sections.

In order to support parallel execution, the different WAMs need to be expanded. Classical implementations using the marker model handle the `&/2` fork-join operator at the abstract machine level: the compiler recognizes `&/2` and compiles it by issuing specific WAM instructions, which are executed by a modified WAM implementation.

These modifications are far from trivial, although they are relatively isolated (e.g., unification instructions are usually not changed, or changed in a generic, uniform way).

Also, the following data areas and stack frames need to be added to each abstract machine [Her86, HG91]:

Goal Stack: A shared area onto which goals that are ready to execute in parallel are pushed. WAMs can pick up goals from other WAM's (or their own) goal stacks. Goal stack entries include a pointer to the environment where the goal was generated and to the code starting the goal execution, plus some additional control information.

Parcall Frames: these stack frames are created for each parallel conjunction and they are utilized for holding the necessary data for coordinating and synchronizing the parallel execution of the goals in the parallel conjunction between the different agents.

Markers: they separate stack sections corresponding to different parallel goals. When a goal is picked up by an agent, an *input marker* is pushed onto the choicepoint stack. Likewise, an *end marker* is pushed when a goal execution ends. These are linked to ensure that backtracking will happen following a logical (i.e., not physical) order.

The following example will clarify the use of these data structures and stack frames in the execution of an and-parallel program.

Example 4 *Figure 2.2 sketches a possible stack layout for the execution of the following predicate* `pred/3`*:*

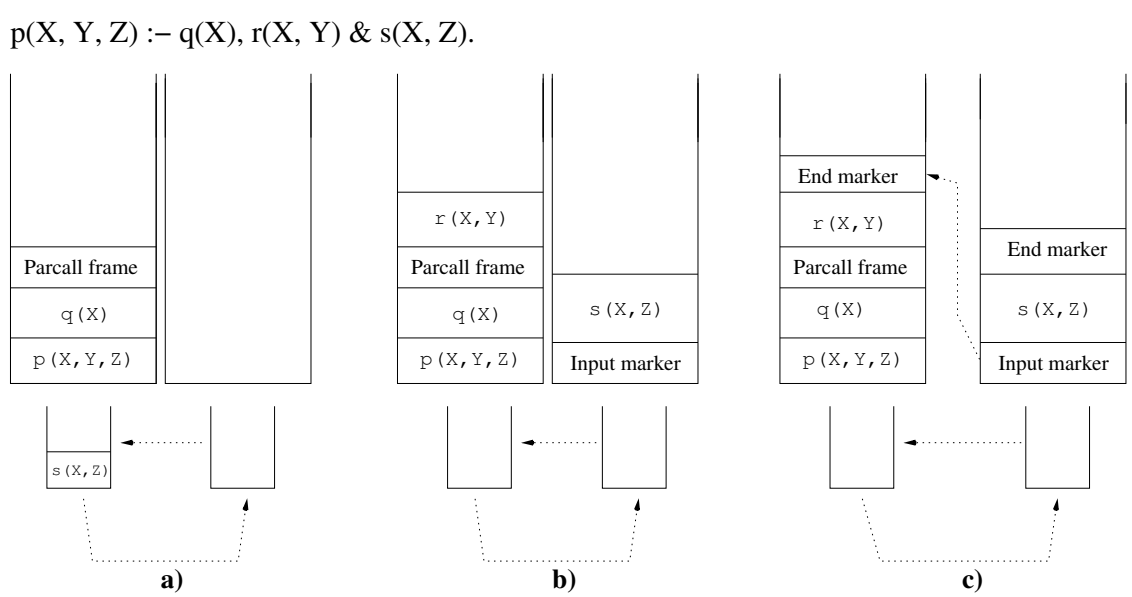


Figure 2.2: Sketch of data structures layout using the marker model.

```
pred(X, Y, Z) :- q(X), r(X, Y) & s(X, Z).
```

with query `pred(X, Y, Z)`*. Assume that* `X` *will be ground after calling* `q(X)`*. Different snapshots of the stack configurations are shown from left to right in Figure 2.2. Note that, in Figure 2.2, parcall frames and markers in the same stack are intermixed. Actual implementations have chosen to place them in different parts of the available data areas. For example, in &ACE parcall frames are not pushed onto the environment stack but on a different stack, and their slots are allocated in the heap, to simplify the memory management.*

When the first WAM executes the parallel conjunction `r(X, Y)` *&* `s(X, Z)`*, it pushes a parcall frame onto its stack and a goal descriptor onto its goal stack for the goal* `s(X, Z)` *(i.e., a pointer to the WAM code that will construct this call in the argument registers and another pointer to the appropriate environment), and it immediately starts executing* `r(X, Y)`*. A second WAM, which is looking for jobs, picks* `s(X, Z)`

up, pushes an input marker into its stack (which references the parcall frame, where data common to all the goals is stored, to be used in case of internal failure*) and constructs and starts executing the goal. Moreover, an end marker is pushed upon completion.*

When the last WAM finishes, it will link the markers (so as to proceed adequately on backtracking and unwinding), and execution will proceed with the continuation of the predicate `pred(X, Y, Z)`*.*

One of the objectives of this thesis is to explore an alternative implementation approach to the marker model, based on raising components to the source language level and keeping at low level only a few selected operations, in order to provide a solution easier to implement, maintain and extend.

Regarding the issue of related work that is closest to the one presented within this thesis, some very early solutions to this problem (e.g., Conery's [Con87] to name one) were based on higher-level constructs, but they were not very efficient.

More recently, [MCN08] has proposed a set of high level multithreading primitives. However, this work is somewhat different in objectives to that of this thesis, concentrating more on providing the user with a flexible multithreading interface, while the objective in this thesis focuses on developing directly and in a simple way a correct and reasonably efficient full implementation of and-parallelism, including the backtracking semantics. As a result, the high-level primitives implemented are more special-purpose and tuned to this concrete objective. Also, performance is a main issue.

Chapter 5 will present this high-level implementation of the execution model that avoids all the modifications to the low-level compiler, and which allows at the same time the execution of unrestricted independent and-parallel programs.

2.6 Summary

Parallelism capabilities have become ubiquitous thanks to the wide availability of multicore systems in the market at a reasonable price. However, the fact that parallelizing programs is a difficult task has renewed the interest in the design and development of support tools, of which parallelizing compilers are a noteworthy instance. More in particular, programs written in functional or logic programming languages can greatly simplify the parallelization process, due to the features of these languages, and several other programming paradigms can benefit from the advances made and the results obtained.

The following chapters will present the work performed on the design, development, implementation and evaluation of automatic parallelization in multiparadigm declarative languages based on a logic programming kernel.

Part II

Functional Notation in LP Systems

Chapter 3

Functions and Lazy Evaluation Support to LP Kernels

This chapter presents the design for a syntactic functional extension, implemented in the Ciao system [BCC+06], which can be implemented in ISO-standard Prolog systems and covers function application, predefined evaluable functors, functional definitions, quoting, and lazy evaluation. The extension is also composable with higher-order features and can be combined with other extensions to ISO-Prolog, such as constraints.

This chapter also highlights the features of the Ciao system which helped to implement this syntactic extension of the language, and presents some performance data on the memory and time overhead of using lazy evaluation with respect to eager evaluation.

3.1 Functional Notation in Ciao

This section presents the main components of the functional notation implemented in Ciao.

3.1.1 Basic Concepts and Notation

The notion of functional notation for logic programming departs in a number of ways from previous proposals. The fundamental one is that functional notation in principle simply provides *syntactic sugar* for defining and using predicates as if they were functions, but they can still retain the power of predicates. In this model, any function definition is in fact defining a predicate, and any predicate can be used as a function. The predicate associated with a function has the same name and one more argument, meant as the place holder for the result of the function. This argument is by default added to the right, i.e., it is the last argument, but this can be changed by using a declaration.

The syntax extensions that have been implemented for functional notation are the following:

Function applications: Any term preceded by the `~` operator is a function application, as can be seen in the goal `write(~arg(1,T))`, which is strictly equivalent to the sequence `arg(1,T,A), write(A)`. To use a predicate argument other than the last as the return argument, a declaration like:

```
:- fun_return functor(~,_,_).
```

can be used, so that `~functor(f,2)` is evaluated to `f(_,_)` (where `functor/3` is the standard ISO-Prolog builtin). This definition of the return argument can also be done on the fly in each invocation as: `~functor(~,f,2)`. Functors

can be declared as *evaluable* (i.e., being in calls in functional syntax) by using the declaration `fun_eval/1`. This allows avoiding the need to use the `~` operator. Thus, "`:- fun_eval arg/2.`" allows writing `write(arg(1,T))` instead of `write(~arg(1,T))` as above. This declaration can also be used to change the default output argument: `:- fun_eval functor(~,_,_).`

Note that all these declarations, as is customary in Ciao, are local to the module where they are included.

Predefined evaluable functors: In addition to functors declared with the declaration `fun_eval/1`, several functors are evaluable, those being:

- The functors used for disjunctive and conditional expressions, `(|)/2` and `(?)/2`. A disjunctive expression has the form `(V1|V2)`, and its value when first evaluated is `V1`, and on backtracking `V2`. A conditional expression has the form `(Cond ? V1)`, or, more commonly, `(Cond ? V1 | V2)`. If the execution of `Cond` as a goal succeeds the return value is `V1`. Otherwise in the first form it causes backtracking, and in the second form its value is `V2`. Due to operator precedences, a nested expression

  `(Cond1 ? V1 | Cond2 ? V2 | V3)`

  is evaluated as `(Cond1 ? V1 | (Cond2 ? V2 | V3))`.

- If the declaration `:- fun_eval arith(true)` is used, all the functors understood by `is/2` are considered evaluable (they will be translated to a call to `is/2`). This is not active by default because several of those functors, like `(-)/2` or `(/)/2`, are traditionally used in Prolog for creating structures. Using `false` instead of `true` the declaration can be disabled.

Functional definitions: A functional definition is composed of one or more functional clauses. A functional clause is written using the binary operator `:=`, as in `opposite(red) := green.`

Functional clauses can also have a body, which is executed before the result value is computed. It can serve as a guard for the clause or to provide the equivalent of where-clauses in functional languages:

```
fact(0) := 1.
fact(N) := N * ~fact(--N) :- N > 0.
```

Note that guards can often be defined more compactly using conditional expressions:

```
fact(N) := N = 0 ? 1
         | N > 0 ? N * ~fact(--N).
```

If the declaration `:- fun_eval defined(true)` is active, the function defined in a functional clause does not need to be preceded by `~` (for example the `fact(--N)` calls above). The translation of functional clauses has the following properties:

- The translation produces *steadfast* predicates [O'K90], that is, output arguments are unified after possible cuts.
- Defining recursive predicates in functional style maintains the tail recursion of the original predicate, thus allowing the usual compiler optimizations.

Quoting functors: Functors (either in functional or predicate clauses) can be prevented from being evaluated by using the `(^)/1` prefix operator (it is read as "quote"), as in `pair(A,B) := ^(A-B)`. Note that this just prevents the evaluation of the principal functor of the enclosed term, not the possible occurrences of other evaluable functors inside.

Scoping: When using function applications inside the goal arguments of meta-predicates, there is an ambiguity as they could be evaluated either in the scope of the outer execution or in the scope of the inner execution. The default behavior is to evaluate function applications in the scope of the outer execution. If they should be evaluated in the inner scope the goal containing the function application needs to be escaped with the `(^^)/1` prefix operator, as in `findall(X, (d(Y), ^^(X = ~f(Y)+1)), L)` (which could also be written as `findall(X, ^^(d(Y), X = ~f(Y)+1), L)`), and whose expansion is `findall(X, (d(Y),f(Y,Z),T is Z+1,X=T), L)`. With no escaping the function application is evaluated in the scope of the outer execution, i.e., `f(Y,Z), T is Z+1, findall(X, (d(Y),X=T), L)`.

Laziness: Lazy evaluation is a program evaluation technique used particularly in functional languages. When using lazy evaluation, an expression is not evaluated as soon as it is assigned, but rather when the evaluator is forced to produce the value of the expression. The `when`, `freeze`, or `block` control primitives present in many modern logic programming systems are more powerful operationally than lazy evaluation. However, they lack the simplicity of use and cleaner semantics of functional lazy evaluation. In this design, a function (or predicate) can be declared as lazy via the declarations:

```
:- lazy fun_eval function_name/N.
```

(or, equivalently in predicate version, "`:- lazy pred_name/M.`", where $M = N + 1$). In order to achieve the intended behavior, the execution of each function declared as lazy is suspended until the return value of the function is needed. Thus, lazy evaluation allows dealing with infinite data structures and also evaluating function arguments only when needed.

Definition of real functions: In the previous scheme, functions are (at least by

default) not forced to provide a single solution for their result, and, furthermore, they can be partial, producing a failure when no solution can be found. A predicate defined as a function can be declared to behave as a real function using the declaration "`:- funct name/N.`". Such predicates are then converted automatically to real functions by adding pruning operators and a number of Ciao assertions [PBH00] which pose (and check) additional restrictions such as determinacy, modedness, etc., so that the semantics will be the same as in traditional functional programming.

3.1.2 Examples

Several examples will illustrate now the use of the syntactic functionality introduced above.

Example 5 *The following example defines a simple unary function* `der(X)` *which returns the derivative of a polynomial arithmetic expression:*

```
der(x)      := 1.
der(C)      := 0                  :- number(C).
der(A + B)  := der(A) + der(B).
der(C * A)  := C * der(A)         :- number(C).
der(x ** N) := N * x ** ~(N - 1) :- integer(N), N > 0.
```

Note that if the directive mentioned is included before which makes arithmetic functors the program would have had to be written in the following (clearly, less pleasant and more obfuscated) way:

```
:- fun_eval(arith(true)).
```

```
der(x)          := 1.
der(C)          := 0                        :- number(C).
der(^(A + B))   := ^(der(A) + der(B)).
der(^(C * A))   := ^(C * der(A))            :- number(C).
der(^(x ** N)) := ^(N * ^(x ** (N - 1))) :- integer(N), N > 0.
```

Both of the previous code fragments translate to the following code:

```
der(x, 1).
der(C, 0) :-
       number(C).
der(A + B, X + Y) :-
       der(A, X),
       der(B, Y).
```

```
der(C * A, C * X) :-
       number(C),
       der(A, X).
der(x ** N, N * x ** N1) :-
       integer(N),
       N > 0,
       N1 is N - 1.
```

Note that in all cases the programmer may use `der/2` *as a function or as a predicate indistinctly.*

Example 6 *Functional notation interacts well with other language extensions. For example, it provides compact and familiar notation for regular types and other properties (assume* `fun_eval` *declarations for them):*

```
color      := red | blue | green.
list       := [] | [_ | list].
list_of(T) := [] | [~T | list_of(T)].
```

which are equivalent to (note the use of higher-order in the third example):

```
color(red).
color(blue).
color(green).

list([]).
list([_|T]) :- list(T).

list_of(_, []).
list_of(T, [X|Xs]) :- T(X), list_of(T, Xs).
```

Such types and properties are then admissible in Ciao-style assertions [PBH00], such as the following, and which can be added to the corresponding definitions and checked by the preprocessor or turned into run-time tests [HPBLG05]:

```
:- pred append/3 :: list * list * list.
:- pred color_value/2 :: list(color) * int.
```

Example 7 *The combination of functional syntax and user-defined operators brings significant flexibility, as can be seen in the following definition of a list concatenation (**append**) operator, defined in an additional Ciao package, named* **functional***:*

```
:- op(600, xfy, (.)).   :- op(650, xfy, (++)).   :- fun_eval (++)/2.
[]   ++ L := L.
X.Xs ++ L := X.(Xs ++ L).
```

*This definition will be compiled exactly to the standard definition of **append** in Prolog (and, thus, will be reversible). The functional syntax and user-defined operators allow*

writing for example: `write("Hello" ++ Spc ++ "world!")` *instead of the equivalent forms* `write( append("Hello", append(Spc, "world!")))` *(if* `append/2` *is defined as evaluable) or* `append(Spc, "world!", T1), append("Hello", T1, T2), write(T2)`.

Example 8 *As another example, an array indexing operator for multi-dimensional arrays can be defined. Assume that arrays are built using nested structures whose main functor is '*`a`*' and whose arities are determined by the specified dimensions, i.e., a two-dimensional array A of dimensions* $[N, M]$ *will be represented by the nested structure* `a(a(`A_{11}`,...,`A_{1M}`), a(`A_{21}`,..,`A_{2M}`), ..., a(`A_{N1}`,...,` A_{NM}`))`, *where* A_{11}`,...` A_{NM} *may be arbitrary terms.*[1] *The following recursive definition defines the property* `array/2` *and also the array access operator* `@`:

```
array([N],A) :-
        functor(A,a,N).
array([N|Ms],A) :-
        functor(A,a,N),
        rows(N,Ms,A).

rows(0,_,_).
rows(N,Ms,A) :-
        N > 0,
        arg(N,A,Arg),
        array(Ms,Arg),
        rows(N-1,Ms,A).

:- op(55, xfx, '@').
:- fun_eval (@)/2.
V@[I]    := ~arg(I,V).  %% Or: V@[] := V.
V@[I|Js] := ~arg(I,V)@Js.
```

This allows writing, e.g., `M = array([2,2]), M@[2,1] = 3` *(which could also be expressed as* `array([2,2])@[2,1] = 3`*), where the call to the* `array` *property gen-*

[1] For simplicity, possible arity limitations are ignored, solved in any case typically by further nesting with logarithmic access time (as in Warren/Pereira's classical library).

erates an empty 2×2 *array* M *and* `M@[2,1] = 3` *puts* 3 *in* $M[2,1]$. *Another example would be:* `A3@[N+1,M] = A1@[N-1,M] + A2@[N,M+2]`.

Example 9 *As a simple example of the use of* lazy evaluation *consider the following definition of a function which returns the (potentially) infinite list of integers starting with a given one:*

```
:- lazy fun_eval nums_from/1.
nums_from(X) := [ X | nums_from(X+1) ].
```

Ciao provides in its standard library the `hiord` package, which supports a form of higher-order untyped logic programming with predicate abstractions [CH99a, Cab04, CHL04]. Predicate abstractions are Ciao's translation to logic programming of the lambda expressions of functional programming: they define unnamed predicates which will be ultimately executed by a higher-order call, unifying its arguments appropriately.[2] A function abstraction is provided as functional syntactic sugar for predicate abstractions:

Predicate abstraction	⇒	Function abstraction
`{''(X,Y) :- p(X,Z), q(Z,Y)}`	⇒	`{''(X) := ~q(~p(X))}`

and the function application is simply defined as syntactic sugar over the predicate application:

[2] A similar concept has been developed independently for Mercury, but their higher-order predicate terms have to be moded.

Predicate application	$\Rightarrow$	Function application
`..., P(X,Y), ...`	$\Rightarrow$	`..., Y = ~P(X), ...`

The combination of this `hiord` package with the `fsyntax` and `lazy` packages (and, optionally, the type inference and checking provided by the Ciao preprocessor [HPBLG05]) basically provide the functionality present in modern functional languages, as well as some of the functionality of higher-order logic programming.

Example 10 *This `map` example illustrates the combination of functional syntax and higher-order logic programming:*

```
:- fun_eval map/2.
map([], _)     := [].
map([X|Xs], P) := [P(X) | map(Xs, P)].
```

With this definition, in the call: `["helloworld", "byeworld"] = map(["hello", "bye"], ++(X)).`, where `(++)/2` corresponds to the above definition of `append`, `X` will be bound to `"world"`, which is the only solution to the equation. Also, when calling:

```
map(L, ++(X), ["hello.", "bye."]).
```

several values for `L` and `X` are returned through backtracking:

```
L = ["hello","bye"],   X = "." ? ;
L = ["hello.","bye."], X = [] ?
```

3.2 Implementation Details

As mentioned previously, certain Ciao features have simplified the proposed extension to handle functional notation. In the following we introduce the features of Ciao that were used and how they were applied in this particular application.

3.2.1 Code Translations in Ciao

Traditionally, Prolog systems have included the possibility of changing the syntax of the source code through the use of the `op/3` builtin/directive. Furthermore, in many Prolog systems it is also possible to define *expansions* of the source code (essentially, a very rich form of "macros") by allowing the user to define (or extend) a predicate typically called `term_expansion/2` [Qui86, CW94]. This is usually how, e.g., definite clause grammars (DCG's) are implemented.

However, these features, in their original form, pose many problems for modular compilation or even for creating sensible standalone executables. First, the definitions of the operators and, specially, expansions are often global, affecting a number of files. Furthermore, it is not possible to determine statically which files are affected, because these features are implemented as a side-effect, rather than a declaration: they become active immediately after being read by the code processor (top-level, compiler, etc.) and remain active from then on. As a result, it is impossible just by looking at a source code file to know if it will be affected by expansions or definitions of operators, which may completely change what the compiler really sees, since those may be activated by the load of other, possibly unrelated, files.

In order to solve these problems, the syntactic extension facilities were redesigned in Ciao, so that it is still possible to define source translations and operators, but

such translations are local to the module or user file defining them [CH00]. Also, these features are implemented in a way that has a well-defined behavior in the context of a standalone compiler, separate compilation, and global analysis (and this behavior is implemented in the Ciao compiler, `ciaoc` [CH99b]). In particular, the `load_compilation_module/1` directive allows separating code that will be used at compilation time (e.g., the code used for program transformations) from code which will be used at run-time. It loads the module defined by its argument *into the compiler*.

In addition, in order to make the task of writing source translations easier, the effects usually achieved through `term_expansion/2` can be obtained in Ciao by means of four different, more specialized directives, which, again, *affect only the current module* and *are (by default) only active at compile-time*.

The proposed functional syntax is implemented in Ciao using these source translations. In particular, `add_sentence_trans/1` and `add_goal_trans/1` directives have been used. A sentence translation is a predicate which will be called by the compiler to possibly convert each *term* (clause, fact, directive, input, etc.) read by the compiler to a new term, which will be used in place of the original term. A goal translation is a predicate which will be called by the compiler to possibly convert each *goal* present in each clause of the current text to another goal which replaces the original one.

Furthermore, this model can be implemented in Prolog systems similarly using the traditional `term_expansion/2` and operator declarations, but having operators and syntactic transformation predicates local to modules is the key to making the approach scalable and amenable to combination with other packages and syntactic extensions in the same application.

3.2.2 Ciao Packages

Packages in Ciao are libraries which define extensions to the language, and have a well defined and repetitive structure. These libraries typically consist of a main source file which defines only some declarations (operator declarations, declarations loading other modules into the compiler or the module using the extension, etc.). This file is meant to be *included* as part of the file using the library, since, because of their local effect, such directives must be part of the code of the module which uses the library. Any auxiliary code needed at compile-time (e.g., translations) is included in a separate module which is to be loaded into the compiler via a `load_compilation_module/1` directive placed in the main file. Also, any auxiliary code to be used at run-time is placed in another module, and the corresponding `use_module` declaration is also placed in the include file.

In this implementation of functional notation in Ciao, two packages have been provided: one for the bare function features without lazy evaluation, and an additional one to provide the lazy evaluation features. The reason for this is that in many cases the lazy evaluation features are not needed and thus the translation procedure is simplified.

3.2.3 Implementation of Functional Extensions in Ciao

In order to perform a translation of the functional definitions, as mentioned above, the `add_sentence_trans/1` directive has been used, which provides a translation procedure in order to transform each functional clause to a predicate clause, adding to the function head the output argument, in order to convert it to the predicate head. This translation procedure also deals with functional applications in heads, as well as with `fun_eval` directives. Furthermore, all function applications are translated

into an internal normal form.

On the other hand, `add_goal_trans/1` directive has been used to provide a translation procedure for dealing with function applications in bodies (which were previously translated into a normal form). The rationale for using a goal translation is that each function application inside a goal will be replaced by a variable, and the goal will be preceded by a call to the predicate which implements the function in order to provide a value for that variable. A simple recursive application of this rule achieves the desired effect.

An additional sentence translation is provided to handle the `lazy` directives. The translation of a lazy function into a predicate is done in two steps. First, the function is converted into a predicate using the procedure sketched above. Then, the resulting predicate is transformed in order to suspend its execution until the value of the output variable is needed. The transformation is explained in terms of the `freeze/1` control primitive that many modern logic programming systems implement quite efficiently [Car87], since it is the most widespread (but obviously `when` [Nai91] or, specially, the more efficient `block` [Car87] declarations can also be used). This transformation renames the original predicate to an internal name and add a *bridge predicate* with the original name which invokes the internal predicate through a call to `freeze/2`, with the last argument (the output of the function) as suspension variable. This will delay the execution of the internal predicate until its result is required, which will be detected as a binding (i.e., demand) of its output variable. The following section will provide a detailed example of the translation of a lazy function. The implementation with `block` is even simpler since no bridge predicate is needed.

As a reference, the main files that are part of the Ciao library packages `fsyntax` are shown below:

```
% fsyntax.pl
:- include(library('fsyntax/ops')).  %% Operator definitions
:- load_compilation_module(library('fsyntax/functionstr')).
:- add_sentence_trans(defunc/3).
:- add_goal_trans(defunc_goal/3).
```

and `lazy` (which will usually be used in conjunction with the first one):

```
% lazy.pl
:- include(library('lazy/ops')). %% Operator definitions
:- use_module(library(freeze)).
:- load_compilation_module(library('lazy/lazytr')).
:- add_sentence_trans(lazy_sentence_translation/3).
```

These files will be *included* in any file that uses the package. The Ciao system source provides the actual detailed code, which follows the our description.

3.2.4 Lazy Functions: an Example

In this section an example of the use of lazy evaluation is presented, which shows how a lazy function is translated by the Ciao package. Figure 3.1 shows in the first row the definition of a lazy function which returns the infinite list of Fibonacci numbers, in the second row its translation into a lazy predicate[3] (by the `fsyntax` package) and in the third row the expansion of that predicate to emulate lazy evaluation (where `fiblist_lazy$$$` stands for a fresh predicate name).

[3]The `:- lazy fun_eval fiblist/0.` declaration is converted into a `:- lazy fiblist/1.` declaration.

```
:- lazy fun_eval fiblist/0.
fiblist := [0, 1 | ~zipWith(+, FibL, ~tail(FibL))]
        :- FibL = fiblist.
```

```
:- lazy fiblist/1.
fiblist([0, 1 | Rest]) :-
          fiblist(FibL),
          tail(FibL, T),
          zipWith(+, FibL, T, Rest).
```

```
fiblist(X) :-
          freeze(X, fiblist_lazy_$$$(X)).

fiblist_lazy_$$$([0, 1 | Rest]) :-
          fiblist(FibL),
          tail(FibL, T),
          zipWith(+, FibL, T, Rest).
```

Figure 3.1: Code translation of a lazy *Fibonacci* function.

In the `fiblist` function defined, any element in the resulting *infinite* list of Fibonacci numbers can be referenced, as, for example, `nth(X, ~fiblist, Value)`. The other functions used in the definition are `tail/2`, which is defined as lazy and returns the tail of a list; `zipWith/3`, which is also defined as lazy and returns a list whose elements are computed by a function having as arguments the successive elements in the lists provided as second and third argument;[4] and `(+)/2` which is defined as by the rule `+(X, Y) := Z :- Z is X + Y`.

Note that the `zipWith/3` function (respectively the `zipWith/4` predicate) is in fact a *higher-order* function (resp. predicate).

[4]It has the same semantics as the `zipWith` function in Haskell.

3.3 Related Work

With respect to the issue of the related work, Lambda Prolog [NM88] offers a highly expressive language with extensive higher-order programming features and lambda-term (pattern) unification. On the other hand it pays in performance the price of being "higher order by default," and is not backwards compatible with traditional Prolog systems. It would be clearly interesting to support pattern unification, but it can be proposed to do it as a further (and optional) extension, and some work is in progress along these lines, but out of the scope of this thesis.

HiLog [CKW93] is a very interesting logic programming system (extending XSB-Prolog) which allows using higher-order syntax, but it does not address the issue of supporting functional syntax or laziness. Functional-logic systems such as Curry or Babel [Han, MNRA89] perform a full integration of functional and logic programming, with higher-order support. On the other hand, their design starts from a lazy functional syntax and semantics, and is strongly typed. However, it may also be interesting to explore supporting narrowing as another optional extension.

Mercury [SHC96] is a programming language which offers functional and higher-order extensions based on Prolog-like syntax, but they are an integral part of the language (as opposed to an optional extension) and, because of the need for type and mode declarations, the design is less appropriate for non strongly-typed, unmoded systems. As mentioned above, in the design type and mode declarations are optional and handled separately through the assertion mechanism. Also, Mercury's language design includes a number restrictions with respect to Prolog-like systems which bring a number of implementation simplifications. In particular, the modedness (no unification) of Mercury programs brings them much closer to the functional case. As a result of these restrictions, Mercury always performs the optimizations pointed out

when discussing the `funct` declaration (or when that type of information is inferred by CiaoPP). However, recent extensions to support constraints [BdlBM+06] recover unification, including the related implementation overheads and mechanisms (such as the trail), and will require analysis for optimization, moving Mercury arguably closer to Ciao in design.

In the case of Oz [HF00], it also allows functional and (a restricted form of) logic programming, and supports higher-order in an untyped setting, but its syntax and semantics are quite different from those of logic programming systems. BIM Prolog offered similar functionality to the `~/2` operator but, again, by default and as a builtin.

3.4 Performance Measurements

Since the functional extensions proposed simply provide a syntactic bridge between functions and predicates, there are only a limited number of performance issues worth discussing. For the case of *real* functions, it is well known that performance gains can be obtained from the knowledge that the corresponding predicate is moded (all input arguments are ground and the "designated output" will be ground on output), determinate, non-failing, etc. [Van94, MCH04]. In Ciao this information can in general (i.e., for any predicate or function) be inferred by the Ciao preprocessor or declared with Ciao assertions [HPBLG05, PBH00]. As mentioned before, for declared "real" (`func`) functions, the corresponding information is added automatically. Some preliminary results on current Ciao performance when this information is available are presented in [MCH04].

In the case of functions that are evaluated lazily, the main goal of the technique presented herein is not really any increase in performance, but achieving new func-

```
:- fun_eval nat/1.
nat(N) :=
     ~take(N, nums_from(0)).

:- lazy fun_eval nums_from/1.
nums_from(X) :=
     [X | nums_from(X+1)].
```

```
:- fun_eval nat/1.
:- fun_eval nats/2.
nat(X) := nats(0, X).
nats(X,Max) := X > Max ? []
            | [X | nats(X+1,Max)].
```

Figure 3.2: Lazy and eager versions of `nat` function.

tionality and convenience through the use of code translations and delay declarations. However, while there have also been some studies of the overhead introduced by delay declarations and their optimization (see, e.g., [MdlBH94]), it is interesting to see how this overhead affects the implementation of lazy evaluation by observing its performance.

For instance, consider the `nat/2` function in Figure 3.2, a simple function which returns a list with the first N natural numbers from an (infinite) list of natural numbers. Function `take/2` in turn returns the list of the first N elements in the input list. This `nat(N)` function cannot be directly executed eagerly due to the infinite list provided by the `nums_from(X)` function, so that, in order to compare time and memory results between lazy and eager evaluation, an equivalent version of that function is provided.

Table 3.1 reflects the time and memory overhead of the lazy evaluation version of `nat(X)` and that of the equivalent version executed eagerly. As a further example, Table 3.2 shows the results for a quicksort function executed lazily in comparison to the eager version of this algorithm. All the results were obtained by averaging ten runs on a medium-loaded Pentium IV Xeon 2.0Ghz, 4Gb of RAM memory, running

	Lazy Evaluation		Eager Evaluation	
List	**Time**	**Heap**	**Time**	**Heap**
10 elements	0.030	1503.2	0.002	491.2
100 elements	0.276	10863.2	0.016	1211.2
1,000 elements	3.584	104463.0	0.149	8411.2
2,000 elements	6.105	208463.2	0.297	16411.2
5,000 elements	17.836	520463.0	0.749	40411.2
10,000 elements	33.698	1040463.0	1.277	80411.2

Table 3.1: Performance results for function `nat/2` (time in ms. and heap sizes in bytes).

	Lazy Evaluation		Eager Evaluation	
List	**Time**	**Heap**	**Time**	**Heap**
10 elements	0.091	3680.0	0.032	1640.0
100 elements	0.946	37420.0	0.322	17090.0
1,000 elements	13.303	459420.0	5.032	253330.0
5,000 elements	58.369	2525990.0	31.291	1600530.0
15,000 elements	229.756	8273340.0	107.193	5436780.0
20,000 elements	311.833	11344800.0	146.160	7395100.0

Table 3.2: Performance results for function `qsort/2` (time in ms. and heap sizes in bytes).

Fedora Core 2.0, with the simple translation of Figure 3.1, and compiled to traditional bytecode (no global optimizations or native code).

It is observable in both tables that there is certainly an impact on the execution time when functions are evaluated lazily, but even with this version the results are quite acceptable if it is taken into account that the execution of the predicate does really suspend.

Related to memory consumption, heap sizes are shown, without garbage collection (in order to observe the raw memory consumption rate). Lazy evaluation implies as

```
:- module(module1, [test/1], [fsyntax, lazy, hiord, actmods]).
:- use_module(library('actmods/webbased_locate')).
:- use_active_module(module2, [squares/2]).

:- fun_eval takeWhile/2.
takeWhile(P, [H|T]) := P(H) ? [H | takeWhile(P, T)]
                            | [].
:- fun_eval test/0.
test := takeWhile(condition, squares).
condition(X) :- X < 10000.
```

```
:- module(module2, [squares/1], [fsyntax, lazy, hiord]).

:- lazy fun_eval squares/0.
:- fun_eval square/1.
:- lazy fun_eval nums_from/1.
squares      := map_lazy(take(1000000, nums_from(0)), square).
square(X)    := X * X.
nums_from(X) := [X | nums_from(X+1)].

:- lazy fun_eval map_lazy/2.
map_lazy([], _)     := [].
map_lazy([X|Xs], P) := [~P(X) | map_lazy(Xs, P)].

:- fun_eval take/2.
take(0, _)     := [].
take(X, [H|T]) := [H | take(X-1, T)] :- X > 0.
```

Figure 3.3: A distributed (active module) application using lazy evaluation.

expected some memory overhead due to the need to copy (freeze) program goals into the heap.

Also, while comparing with standard lazy functional programming implementations is beyond the scope of this thesis, some simple tests done for sanity check purposes (with HUGS) show that the results are comparable, the implementation being for example slower on `nat` but faster on `qsort`, presumably due to the different optimizations being performed by the compilers.

An example when lazy evaluation can be a better option than eager evaluation in terms of performance (and not only convenience) can be found in a concurrent or distributed system environment (such as, e.g., [CH02]), and in the case of Ciao also within the active modules framework [BCC+06, CH95].

The example in Figure 3.3 uses a function, defined in an active module, which returns a big amount of data. Function `test/0` in module `module1` needs to execute function `squares/1`, in (active, i.e., remote) module `module2`, which will return a very long list (which could be infinite). If `squares/1` were executed eagerly then the entire list would be returned, to immediately execute the `takeWhile/2` function with the entire list. `takeWhile/2` returns the first elements of a (possibly infinite) list while the specified condition is true.

However, creating the entire initial list is very wasteful in terms of time and memory requirements. In order to solve this problem, the `squares/1` function could be moved to module `module1` and be merged with the `takeWhile/2` function (or, also, they could exchange a size parameter). However, rearranging the program is not always possible and it may also perhaps complicate other aspects of the overall design.

If, on the other hand, the `squares/1` function is evaluated lazily, it is possible to keep the definitions unchanged and in different modules, so that there will be a smaller time and memory penalty for generating and storing the intermediate result. As more values are needed by the `takeWhile/2` function, more values in the list returned by the `squares/1` function are built (in this example, only while the new generated value is less than $10,000$), considerably reducing the time and memory consumption.

3.5 Summary

We have presented a syntactic approach to allow the usage of functions in a logic programming kernel. This proposal and its implementation offer a combination of features which make it interesting in itself. More concretely, functions can be limited or retain the power of predicates, any predicate can be called through functional syntax, and lazy evaluation is supported both for functions and predicates. Furthermore, the functional syntax can be combined with numerous syntactic and semantic extensions such as higher-order, assertions, records, constraints, objects, persistence, other control rules, etc., without any modification to the compiler or abstract machine. This extension to the language has been implemented in the Ciao system.

Part III

Unrestricted Independent And-Parallelism

Chapter 4

Annotation Algorithms for Unrestricted IAP

This chapter provides and evaluates two different algorithms for the annotation process of unrestricted independent and-parallelism. The first algorithm does not need to preserve the order of the solutions with respect to the sequential program, and the second algorithm must respect this order. Also, the total correctness of both algorithms will be proved. Additionally, these annotation algorithms will make use of determinacy information in order to obtain better performance results.

4.1 Motivation and Related Work

This section presents a comparison between restricted and unrestricted parallelization, showing how the unrestricted parallelization solves the limitation of the restricted one.

4.1.1 *Fork-Join*-Style Parallelization

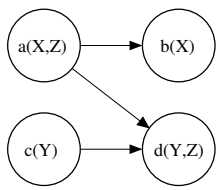
In order to show the limitations of the fork-join operator `&/2`, consider the clause for `p/1` in Figure 4.1(a) and the parallelization depicted in Figure 4.2(b). The parallel fork-join operator `&/2` binds more tightly than the comma. Thus, the expression "`a(X,Z) & b(X), c(Y) & d(Y,Z)`" means that literals `a/2` and `b/1` can be safely executed in parallel. Execution can continue only when both `a/2` and `b/1` have successfully finished. At that point `c/1` and `d/2` can start parallel execution.

As mentioned in Chapter 2, whether two goals can in fact be parallelized depends on whether they are independent, under some notion of independence. As also mentioned previously, it will be assumed that this information is obtained from local and global data-flow analysis. For the sake of simplicity, it will also be assumed in the rest of the chapter that all the dependencies are unconditional —i.e., conditional dependencies are assumed to be always false, reducing the use of run-time checks (and if-then-else constructions), which, as also mentioned before, is a reasonable compromise between parallelism and run-time overhead. For this example, assume that the dependencies detected between the literals in predicate `p/3` are `dep(a,b)`, `dep(a,d)`, `dep(c,d)`, where `dep(X,Y)` denotes that `Y` depends on (and therefore must be executed before) `X`. The resulting conditional dependency graph for predicate `p/3` is shown in Figure 4.1(b). In such a graph the vertices V correspond to the literals of the clause and there exists an edge between two literals L_i and L_j in E if $ind(L_i, L_j) \neq true$ (i.e., $dep(L_i, L_j) = true$, and thus literal L_i has to be completed before literal L_j), where *ind* is the *notion of independence*. This approach is thus parametric on the notion of independence.

Given such a graph, several annotations are possible. As an example, Figure 4.2 shows two of the possible annotations for the running example. There are actually other possible parallelizations, as for instance `p :- a(X, Z), b(X) & c(Y), d(Y,`

```
p(X,Y,Z) :-
      a(X,Z),
      b(X),
      c(Y),
      d(Y,Z).
```

(a) Predicate *p/3*.



(b) Dependency graph.

Figure 4.1: Predicate `p/3` and its associated dependency graph.

```
p(X, Y, Z):-
        (a(X, Z), b(X)) & c(Y),
        d(Y, Z).
```

(a) *fj1*: Order-preserving

```
p(X, Y, Z):-
        a(X, Z) & c(Y),
        b(X) & d(Y, Z).
```

(b) *fj2*: Non-order-preserving

Figure 4.2: Fork-join annotations for predicate `p/3`.

`Z)`, which has been left out of Figure 4.2 for brevity (and because they do not add anything to the discussion since they do not change the comparisons made in Section 4.1.2). Some goals appear switched with respect to their order in the sequential clause. This respects the dependencies in Figure 4.1(b), which reflects a valid notion of parallelism (i.e., if solution order is not important). If additional ordering requirements are needed (due to, e.g., side effects or impurity), these can be included as additional edges in the graph.

It is possible in some cases to statically decide (or, at least, approximate) whether some annotation is better than some other, for example by using the number of goals annotated for parallelism in a clause or, more interestingly, by using information regarding the expected runtime of goals (see, e.g., [DLH90, DL91, DL93, DLGHL94, LGHD96, DLGHL97, DLGH97, MLGCH08] and their references). However, even if the actual costs are known, finding an optimal scheduling solution is a computationally expensive combinatorial problem [MBdlBH99] and, in practice, since there are

many decisions to make which multiply the number of possible annotations, annotators use heuristics which may be more or less appropriate in concrete cases. Thus, these heuristics are part of the important differences between annotators.

For example, consider the annotations in Figure 4.2. First, it should be noted that none of these annotations (or, in fact, any of the other possible alternatives using `&/2`), fully exploit all the parallelism available in Figure 4.1(b): Figure 4.2(a) misses the possible parallelism between literals `b/1` and `d/2`, while Figure 4.2(b) misses the parallelism between literals `b/1` and `c/1`. As for which of these two parallelizations is better, a clearly meaningful measure of their quality is how long each of them takes to execute. These times will be termed T_{fj1} and T_{fj2}, for Figures 4.2(a) and 4.2(b), respectively. This length depends on the execution times of the goals involved (i.e., T_a, T_b, T_c, T_d), which are assumed to be non-zero. T_{fj1} and T_{fj2} are:

$$T_{fj1} = \max(T_a + T_b,\ T_c) + T_d \tag{4.1}$$

$$T_{fj2} = \max(T_a,\ T_c) + \max(T_b,\ T_d) \tag{4.2}$$

Comparing the quality of the annotations in Figure 4.2(a) and Figure 4.2(b) boils down to finding out whether it is possible to show that $T_{fj1} < T_{fj2}$ or the other way around.

It turns out that these execution time expressions are non-comparable, since there are solutions for both orderings, so none of them is definitely better than the other one:

- $T_{fj1} < T_{fj2}$ holds if, for example, $T_a + T_b < T_c$, $T_d < T_b$, and then $T_{fj2} = T_b + T_c$, $T_{fj1} = T_d + T_c$, and
- $T_{fj2} < T_{fj1}$ holds if, for example, $T_c \leq T_a$, $T_d \leq T_b$, and then $T_{fj1} = T_a + T_b + T_d$,

$$T_{fj2} = T_a + T_b.$$

While it is not possible to show that one of these two annotations is better in general than the other, the reasoning used will be instrumental to show that the annotations proposed in Section 4.1.2 are better than any of those possible with the `&/2` fork-join operator. It is well-known that the class of graphs the fork-join operator can express directly (i.e., dependency graphs with a nested fork-join structure) is a subset of that which can possibly be generated from programs [MBdlBH99]. This makes parallelism opportunities to be inevitably lost in cases with a complex enough structure (and that one in Figure 4.1(b) is an example). Furthermore, inter-procedural parallelism (i.e., parallel conjunctions which span literals in different predicates) cannot be exploited easily with the `&/2` operator without complex program transformations.

4.1.2 Parallelization with Finer Goal-Level Operators

Two motivations justify the use of these operators instead of the restricted operator `&/2`:

- Their implementation is easier to devise and maintain than the monolithic `&/2` [CCH08b].
- The operators `&>/2` and `<&/1` allow more freedom to the annotator (and to the programmer, if parallel code is written by hand) to express data dependencies and, therefore, to extract more potential parallelism.

This last point will be now illustrated in this chapter, since the former will be addressed in Chapter 5. Note that the `&>/2` and `<&/1` operators do not replace the

fork-join operator `&/2` at the language level due to its conciseness in cases in which no extra parallelism can be exploited with `&>/2` and `<&/1`.

Figure 4.3 shows an annotation of the running example using the unrestricted operators. Note that this code allows executing in parallel `a/2` with `c/1`, `b/1` with `c/1`, and `b/1` with `d/2`. As in Equations (4.1) and (4.2), the execution time of `p/3` is based on that of the goals in its body and can be worked out as shown in Figure 4.4. In that figure, the clause goals appear at the left and the time needed to execute up to just after each of these body goals appears at the right. Time is relative to that of the neck traversal. T_n (with $n \in \{a, b, c, d\}$) denotes the execution time of the respective goals. The primitives `&>`/2 and `<&`/1 are, for simplicity, assumed to take no time. Then, T_7, the total time taken by the clause, can be solved as a function of the length of the goals:

$$\begin{aligned} T_7 &= \max(T_6,\ T_3 + T_b) \\ &= \max(T_5 + T_d,\ T_2 + T_a + T_b) \\ &= \max(\max(T_4,\ T_1 + T_c) + T_d,\ T_a + T_b) \\ &= \max(\max(T_a,\ T_c) + T_d,\ T_a + T_b) \end{aligned}$$

Thus, the execution time of `p/3` is:

$$T_{dep} = \max(\max(T_a,\ T_c) + T_d,\ T_a + T_b) \tag{4.3}$$

While Equations (4.1) and (4.2) were incomparable, an analysis of Equation (4.3) compared with these two other equations shows that:[1]

[1]The details of the analysis are not shown here. However, it is trivial to perform using a CLP system, as shown in Appendix A.

```
p(X, Y, Z) :-
        c(Y) &> Hc,
        a(X, Z),
        b(X) &> Hb,
        Hc <&,
        d(Y, Z),
        Hb <&.
```

Figure 4.3: Unrestrictedly annotated clause of predicate `p/3`.

- $T_{dep} < T_{fj1}$ holds if, for example, $T_c \leq T_a$, and then $T_{dep} = \max(T_a+T_b, T_a+T_d)$, $T_{fj1} = T_a + T_b + T_d$.
- $T_{dep} < T_{fj2}$ holds if, for example, $T_a < T_c$, $T_c + T_d \leq T_a + T_b$, and then $T_{dep} = T_a + T_b$, $T_{fj2} > T_a + T_b$.
- $T_{dep} = T_{fj1}$ holds if, for example, $T_b < T_c$, $T_a + T_b = T_c$, and then $T_{dep} = T_c + T_d = T_{fj1}$.
- $T_{dep} = T_{fj2}$ holds if, for example, $T_d \leq T_b$, $T_c \leq T_a$, and then $T_{dep} = T_a + T_b = T_{fj2}$.

Additionally, there is **no** case in which $T_{dep} > T_{fj1}$ or $T_{dep} > T_{fj2}$. Therefore, the annotations in Figure 4.2 (and, in fact, any other possible annotation for this clause using `&>/2`) are, with the aforementioned assumptions regarding the execution time of `&>/2`, `<&/1`, and `&/2`, worse than the annotation in Figure 4.3. The *non-fork-join* annotation is, therefore, a better option than any of the other *fork-join* annotations.

In addition to the basic unrestricted operators, other specialized versions can be defined and implemented in order to increase performance by adapting better to some particular cases. For example, it appears interesting to introduce variants for the very relevant and frequent case of deterministic goals, in which backward

```
p(X, Y, Z):-
    c(Y) &> Hc,
    a(X, Z),
    b(X) &> Hb,
    Hc <&,
    d(Y, Z),
    Hb <&.
```

$$
\begin{aligned}
T_1 &= 0 \\
T_2 &= T_1 \\
T_3 &= T_2 + T_a \\
T_4 &= T_3 \\
T_5 &= \max(T_4,\ T_1 + T_c) \\
T_6 &= T_5 + T_d \\
T_7 &= \max(T_6,\ T_3 + T_b)
\end{aligned}
$$

Figure 4.4: Deduction of execution time for unrestricted parallelization of predicate `p/3`.

execution does not need to be performed —and, therefore, forward execution does not need to be *prepared* for backtracking. For this purpose, two additional operators are proposed: `&!>/2` and `<&!/1`, which incur the overhead of having to prepare the execution data structures to cope with the possibility of backtracking. This has previously been shown to result in a significant efficiency increase in the underlying machinery [Her86, PGT+96].

4.2 The UUDG and UOUDG Algorithms

This section will present two concrete algorithms which generate code annotated for unrestricted independent and-parallelism (as in Figure 4.3) starting from sequential code. The proposed algorithms process one clause at a time and work on a directed acyclic dependency graph, where nodes are, in general, associated with a sequence

of clause body goals, to be sequentially executed.[2] It is required that literals which are lexically identical to be distinguished by, e.g., attaching a unique identifier to them. This is necessary in order not to lose information when building sets of nodes, needed for the algorithms which will be presented later.

The idea behind these algorithms is to publish (i.e., to make available) goals for parallel execution as soon as possible and to delay "importing" their bindings (i.e., issuing joins) as much as possible —but always respecting the dependencies in the graph (as in Figure 4.1(b)). Intuitively, this should maximize the number of goals available for parallel execution, and preserve the order of the solutions, if required.

The external interface of the annotation process is shown in the algorithm in Figure 4.5. The first argument is the dependency graph associated to the clause to be parallelized. The second argument corresponds to a boolean which determines whether an order-preserving annotation is to be performed or not, and which is used to decide which procedure (i.e., which of the algorithms Figures 4.6 and 4.9) should be called. The last argument contains determinacy information on the clause literals, needed to reorganize body goals (if this is the selected option) or, in any case, to generate deterministic parallel annotations.

Note that, as mentioned in Section 4.1.1, only unconditional parallelism will be considered, for simplicity and also because, as mentioned before, it has shown to be a good compromise and effective as a default strategy in practice. However, the algorithms can be easily adapted to deal with conditional parallelism without too much effort.

In what follows, the dependency graph G will be denoted as the pair $G = (V, E)$ where V is the set of vertices or nodes, and E is the set of edges. $G|_U$ will denote

[2] As we will see later, it is possible to *group* goals in the output parallelized clause in order to optimize execution time.

the subgraph $(U, E|_U)$ of G where $E|_U \in E$ has only the edges in E connecting those nodes in U. Set difference is defined as usual as $A \setminus B = \{x \mid x \in A,\ x \notin B\}$. The relation $(x \rightsquigarrow y)_E$ expresses that there is a path from x to y using edges in E. $\texttt{incoming}(\texttt{v}, \texttt{E}) = \{u \mid (u, v) \in E\}$ denotes the set of nodes which are connected to some particular node v. The function $\texttt{min\_card}(\texttt{S}) = \min_{s \in S} |s|$ returns the size of the smallest set in the set of sets S.

In order to keep track of the order of the solutions, it is assumed that there exists a relation $\prec$ on the literals L_i of the body of every clause H :- $L_1, L_2, \ldots, L_{k-1}, L_k$ such that $L_i \prec L_j$ if and only if $i < j$. Additionally, it is assumed that there exists a partial function $\texttt{pred}(L)$ which is defined as $\texttt{pred}(L_{i+1}) = L_i$, i.e., it returns the literal at the left of some other literal in a clause.

It is also assumed that $\prec$ and $\texttt{pred}(L)$ are suitably extended, in the straightforward way, to the nodes of the dependency graph (recall that nodes can have associated sequences of adjacent literals in the original clause). Note also that graph edges must respect the $\prec$ relation: $(u, v) \in E \Rightarrow u \prec v$, since the graph would have been incorrectly generated otherwise.

Both algorithms use the function $\texttt{get\_literals}(v)$, which returns the set of literals of the original clause associated to the node v. The input dependency graph $G = (V, E)$ used as input in Algorithm 4.5 is initially built so that $\forall v_i \in V,\ \texttt{get\_literals}(v_i) = \{L_i\}$. In addition, $\texttt{set\_literals}(u, S)$ associates to the node $u \in V$ the set of literals which are associated to the nodes in S, i.e., after $\texttt{set\_literals}(u, S)$ the value returned by $\texttt{get\_literals}(u)$ is $\bigcup_{v \in S} \texttt{get\_literals}(v)$.

Algorithm: `UnrestrictedAnnotation`(G, *order*, I_D)
Input: **(1)** A directed acyclic dependency graph $G = (V, E)$.
(2) The boolean value *order*.
(3) Determinacy information I_D for the literals of the clause.
Output: A clause annotated for unrestricted independent and-parallel execution.
begin
  if *order* **then**
    $Exp \leftarrow$ UOUDG(G, I_D);
  else
    $Exp \leftarrow$ UUDG(G, I_D);
  end
  return Exp
end

Figure 4.5: Entry point to the annotation algorithms.

4.2.1 Non Order-Preserving Annotation: the UUDG Algorithm

Figure 4.6 presents an algorithm that parallelizes a clause, represented as an (acyclic) directed dependency graph.

At every iteration step, new nodes in the graph are selected to be published, joined or executed sequentially. Subsequent iterations proceed with a simplified graph in which the literals which have been joined or executed sequentially, together with their outgoing edges, have been removed. The set of goals which have already been published is kept in a separate parameter in order not to schedule goals for parallel execution more than once.

In order to not lose parallelism, sequences of goals which have to be necessarily run sequentially are collapsed in a single node at the beginning of each iteration. These sequences are characterized because there exists a path from every node in the sequence to some successor literal in the clause and there are no incoming edges

Algorithm: UUDG(G_i, I_D)
Input: **(1)** A directed acyclic graph $G_i = (V_i, E_i)$.
(2) Determinacy information.
Output: A parallelized clause $\texttt{expr}_{G_i}$.
begin
$\texttt{expr}_{G_i} \leftarrow (\texttt{true})$;
$Pub \leftarrow \emptyset$;
$G = (V, E) \leftarrow G_i$;
while $V \neq \emptyset$ **do**
$G \leftarrow \texttt{group\_nonord}(G,\ Pub)$; $Indep \leftarrow \{v \mid v \in V,\ \texttt{incoming}(v, E) = \emptyset\}$;
$Dep \leftarrow \{I_v \mid v \in V,\ I_v = \texttt{incoming}(v, E),\ I_v \neq \emptyset,\ I_v \subseteq Indep\}$;
if $Dep = \emptyset$ **then**
$SS \leftarrow \emptyset$;
$Join \leftarrow V$;
else
$SS \leftarrow \{I \mid I \in Dep,\ |I| = \texttt{min\_card}(Dep)\}$;
$Join \leftarrow s$ s.t. $s \in SS$; `/* s any element from SS */`
end
if $(Join \cap (Indep \setminus Pub)) = \emptyset$ **then**
$Seq \leftarrow \emptyset$;
else
$Seq \leftarrow \{v\}$ s.t. $v \in (Join \cap (Indep \setminus Pub))$; `/* v any element */`
end
$Fork \leftarrow Indep \setminus (Pub \cup Seq)$;
$Join \leftarrow Join \setminus Seq$;
$Pub \leftarrow Pub \cup (\bigcup_{v \in Fork} \texttt{get\_literals}(v)) \cup \texttt{get\_literals}(u)$ s.t. $u \in Seq$;
$G \leftarrow G|_{(V \setminus Join) \setminus Seq}$;
$\texttt{expr}_{G_i} \leftarrow (\texttt{expr}_{G_i}, \texttt{gen\_body\_nonord}(Fork, Seq, Join, I_D))$;
end
return $\texttt{expr}_{G_i}$;
end

Figure 4.6: UUDG annotation algorithm.

starting in a node outside the sequence. The intuition behind this is to detect when the sequence as a whole can be started as a *compound* parallel goal. This is performed by running the function $\texttt{group\_nonord}(G, P)$, shown in Figure 4.7.

Two sets are key in each iteration:

***Indep*,** which contains the *sources* (i.e., all vertices without incoming edges in the current graph, which can therefore be published), and

***Dep*,** which contains sets of vertices $I_v \subseteq Indep$ s.t. for each non-source v which can only be reached from sources, I_v is the set of sources on which the literals in v depend. I.e., I_v is the set of vertices to be joined before the literals in v can start.

If there are no sets of vertices in *Dep*, then all the sequences of literals that remain in the graph are independent, and thus they can all be published and joined up. Otherwise, a set of nodes needs to be chosen from *Dep* in order to wait for the result of their associated literals to be ready. The choice within that set is made by selecting, among the sets of goals which can be joined at every moment, the one with the lowest cardinality (using `min_card(S)`), thus postponing the rest of the joins as much as possible, in order to exploit more parallelism.

Note that a random selection from a set is made at two different points (marked with comments stating `... any element...`). Data regarding, e.g., the relative expected run-time of goals would allow us to take a more informed decision and therefore to precompute a perhaps better scheduling. Since this information is not used here, any available goal to join / execute sequentially is just selected.

It is possible for a literal to be scheduled to be forked and then immediately joined. In order to detect these situations, which in practice would cause an unnecessary overhead, literals (in *Seq*, only one at a time) to which this applies are detected, and they are not taken into account for the set of *Fork*ed nodes and they are removed from the set of the *Join*ed nodes. Other literals in the same situation will eventually be selected in subsequent iterations.

Algorithm: `group_nonord`(G, Pub)
Input: **(1)** A nonempty directed acyclic graph $G = (V, E)$.
(2) A set of goals already forked.
Output: A compacted directed acyclic graph
begin
forall $v \in V$ s.t. `get_literals`(v) $\not\subseteq Pub$ **and** `incoming`(v, E) $= \emptyset$ **do**
$NewV \leftarrow \{v\}$;
$DS \leftarrow \{u \mid u \in V,\ u \notin NewV,\ w \in NewV,\ (w, u) \in E\}$;
while $DS \neq \emptyset$ **do**
$NewV' \leftarrow NewV \cup DS$;
if $(\forall \{v_i, v_j\} \subseteq NewV',\ (v_i \rightsquigarrow v_j)_E\ \vee\ (v_j \rightsquigarrow v_i)_E)$ **and**
$(\forall e = (v_k, v_l) \in E,\ v_k \notin NewV' \Rightarrow v_l \notin NewV')$ **then**
$NewV \leftarrow NewV'$;
else
break;
$DS \leftarrow \{u \mid u \in V,\ u \notin NewV,\ w \in NewV,\ (w, u) \in E\}$;
end
`set_literals`(v, $NewV$);
$G \leftarrow G|_{(V \setminus (NewV \setminus \{v\}))}$;
end
return G;
end

Figure 4.7: Nonorder-preserving grouping of nodes.

The UUDG algorithm then continues outputting a parallel expression generated by the function `gen_body_nonord`(F, S, J, I_D), shown in Figure 4.8, composed with the parallelization of a simplified graph, generated by an iterative call. It makes use of the definition `seq`(S), which sequentializes the literals in the set S preserving their order in the original clause. Those literals associated to the node in *Seq*, if any, are annotated after all literals in *Fork* have been published for parallel execution, in order to exploit all the detected parallelism.

The function `gen_body_nonord`(F, S, J, I_D) makes use of determinism information, provided by the auxiliary predicate `det`(n, I_D), which is true when the literals

associated to a particular node n of the graph are deterministic or not, as follows:

- When the literals are known to have exactly one solution, the specialized versions of the unrestricted operators, `&!>/2` and `<&!/1` which do not perform bookkeeping for backtracking (always complex in parallel implementations), can be used and are thus more efficient.
- Additionally, since it is possible to switch goals around, deterministic goals will be forked first in order to try to minimize relaunching goals which are likely to be executed in parallel.

Example 11 *This example will show how having determinism information can help in achieving a better parallelization. Suppose the following predicate* `p/4`*:*

```
p(A,B,C,D) :- a(A), b(B), c(C), d(D).
```

where literals `a/1` *and* `d/1` *are deterministic and literals* `b/1` *and* `c/1` *are nondeterministic. Let us assume that these four literals are independent. Then, the parallelization obtained by the* UUDG *algorithm is:*

```
p(A,B,C,D) :-
      a(A) &!> Ha, d(D) &!> Hd, b(B) &> Hb,
      c(C),
      Ha <&!, Hd <&!, Hb <&.
```

Thus, the solutions of literal `b/1` *will not need to be recomputed each time a previous parallel goal returns a new solution, since literals* `a/1` *and* `d/1` *are deterministic, i.e., only one solution will be obtained.*

Algorithm: `gen_body_nonord`(*Fork*, *Seq*, *Join*, I_D)
Input: **(1)** A set of vertices to be forked.
(2) A set of vertices to be sequentialized.
(3) A set of vertices to be joined.
(4) Determinacy information.
Output: A parallelized sequence of literals *Exp*.
begin
Exp ← (`true`);
ForkDet ← $\{n \mid n \in Fork,\ det(n, I_D)\}$; *ForkNonDet* ← *Fork* \ *ForkDet*;
JoinDet ← $\{n \mid n \in Join,\ det(n, I_D)\}$; *JoinNonDet* ← *Join* \ *JoinDet*;
forall $v_i \in ForkDet$ **do**
Exp ← (*Exp*, `seq(get_literals(v`$_i$`))` `&!>` H_{v_i});
end
forall $v_i \in ForkNonDet$ **do**
Exp ← (*Exp*, `seq(get_literals(v`$_i$`))` `&>` H_{v_i});
end
if *Seq* = $\{v\}$ **then**
Exp ← (*Exp*, `seq(get_literals(v))`);
end
forall $v_i \in JoinDet$ **do**
Exp ← (*Exp*, H_{v_i} `<&!`);
end
forall $v_i \in JoinNonDet$ **do**
Exp ← (*Exp*, H_{v_i} `<&`);
end
return *Exp*;
end

Figure 4.8: Nonorder-preserving generation of a parallel body.

Determinism information can often be automatically inferred by tools such as the abstract interpretation-based determinism analyzer included in CiaoPP [LGBH05], and can be provided as input to the annotators presented in this chapter. This information can alternatively be stated by the programmer via assertions [HPBLG05].

4.2.2 Correctness Proof of the UUDG Algorithm

This section presents the total correctness proof of the UUDG algorithm. The total correctness of the `group_nonord`(G, P) function will be proved first.

Lemma 1 *Assume a non-empty graph $G = (V, E)$ and a particular set of nodes $NewV \subseteq V$ obtained by the function* ***group_nonord****. Then, no parallelism is lost when the nodes in $NewV$ are grouped into the source $v \in NewV$.*

Proof 1 *Let us prove this by induction on V.*

- *Base case: if $|V| = 1$ then $E = \emptyset$ and trivially no parallelism can be lost. Note that $V \neq \emptyset$ because of the precondition in function* ***group_nonord****.*
- *Induction hypothesis: assume that nodes in $NewV$ are mutually dependent.*
- *Inductive step: let us prove this by contradiction. Assume that some new nodes in DS are added to $NewV$. Then, there are three cases in which parallelism may be lost:*
  1. *If $NewV$ contains now two sources u, w, which would then not be candidates to be independently scheduled for parallel execution. However, this is not possible because of the definition of DS.*
  2. *Assume that $\exists u, w \in NewV$ s.t. $(u \not\rightsquigarrow w)_E \wedge (w \not\rightsquigarrow u)_E$ —i.e., two nodes in the newly grouped set are independent. Then the possible parallelism between u and w will be lost. However, because of the first condition in the if-structure, $x \in NewV$ if $\forall \{x, y\} \subseteq NewV$, $(x \rightsquigarrow y)_E \vee (y \rightsquigarrow x)_E$, which means that $\nexists \{x, y\} \in NewV$ s.t. $(x \not\rightsquigarrow y)_E \wedge (y \not\rightsquigarrow x)_E$, and that leads to a contradiction.*

3. *Assume that* $\exists u \in NewV$, $\exists w \notin NewV$ *s.t.* $\exists(w,u) \in E \vee \exists(u,w) \in E$. *Then the possible parallelism between* w *and the nodes* $v_i \in NewV$ *s.t.* $(w \not\rightsquigarrow v_i)_E \ \wedge \ (v_i \not\rightsquigarrow w)_E$ *will be lost. If* $\nexists v_i \in NewV$ *s.t.* $(w \not\rightsquigarrow v_i)_E \ \wedge \ (v_i \not\rightsquigarrow w)_E$ *then, because of the first condition in the if-structure,* $w \in NewV$ *and that leads to a contradiction. Otherwise,*

   (a) *If* $(w,u) \in E$ *then, because of the second condition in the if-structure,* $u \notin NewV$, *which leads to a contradiction.*

   (b) *If* $(u,w) \in E$ *then* $\exists(u,v_i) \in E$ *s.t.* $(w \not\rightsquigarrow v_i)_E \ \wedge \ (v_i \not\rightsquigarrow w)_E$. *Because of the first condition in the if-structure, since* $w \notin NewV$ *then* $v_i \notin NewV$, *and that leads to a contradiction.*

Thus no parallelism is lost.

Lemma 2 (Partial Correctness of Algorithm 4.7) *The function* `group_nonord`, *when called with input graph* $G = (V,E)$ *and with output graph* $G_f = (V_f, E_f)$, *is partially correct with respect to the precondition* $\{|V| > 0\}$ *and the postcondition* $\{P_1; P_2; P_3\}$, *where:*

1. $P_1 \equiv \bigcup_{v \in V_f}$ `get_literals`$(v) = V$.
2. $P_2 \equiv \bigcap_{v \in V_f}$ `get_literals`$(v) = \emptyset$.
3. $P_3 \equiv$ *no parallelism is lost when constructing* G_f.

Proof 2 *Starting with values that make the precondition* true, *for each source* $v \in G$ *which has not been published yet, Lemma 1 ensures that no parallelism is lost for any of the groups of literals created, and therefore no parallelism is lost when building* G_f *(property* P_3*). Moreover, only nodes that have been grouped into a source are*

removed from the graph, and thus no nodes are missing in G_f. In addition, G is simplified in each iteration, so there will be no repeated nodes in G_f. Therefore, the postcondition is true *and thus the function* **`group_nonord`** *is partially correct.*

Lemma 3 (Termination of Algorithm 4.7) *The function* **`group_nonord`** *terminates.*

Proof 3 *For the inner loop, the set of natural numbers $\mathbb{N}$is chosen as set with strict well-founded ordering $<$. Let $|V \setminus NewV|$ be the termination expression. By definition, $|V \setminus NewV| \in \mathbb{N}$. $|V \setminus NewV|$ reduces its value in each iteration with respect to $<$ if $NewV \neq \emptyset$, so $NewV' \neq \emptyset$, and then $DS \neq \emptyset$, which is the exit condition of the loop. Since the outer loop simply consists of a single execution of the inner loop over each element of a finite set, the algorithm terminates.*

Theorem 1 (Total Correctness of Algorithm 4.7) *The function* **`group_nonord`** *is totally correct.*

Proof 4 *Lemma 2 states that the function* **`group_nonord`** *is partially correct, and Lemma 3 states that it terminates, so the function* **`group_nonord`** *is totally correct.*

Before proceeding with the total correctness proof of the UUDG algorithm, the equivalence class of graphs with respect to transitive edges will be introduced:

Definition 8 (Equivalence class of graphs w.r.t. $\equiv_t$) *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two different dependency graphs. Then, $G_1 \equiv_t G_2 \Leftrightarrow (V_1 = V_2) \wedge (\forall e = (a, b) \in E_1,\ e \notin E_2 \Rightarrow (a \rightsquigarrow b)_{E_2})$. I.e., direct dependencies in one of the graph appear in the other one too, but maybe traversing several edges.*

Lemma 4 *In any iteration of the* UUDG *algorithm over a graph* $G = (V, E)$, Fork, Seq *and* Join *are composed only by sources.*

Proof 5

1. Join $= V$ *if* $E = \emptyset$, *or that* Join $\in$ Dep. *In this case,* $\forall S \in Dep,\ S \subseteq$ Indep, *then* Join $\subseteq$ Indep. *Since* Indep *is composed only by sources then* Join *is so as well.*

2. Seq $= \emptyset$, *or that* Seq $= \{v\}$ s.t. $v \in$ (Join $\cap$ (Indep $\setminus$ Pub)) $\subseteq$ Join. *Because of the result above,* Join $\subseteq$ Indep *and thus* Seq *is composed only by sources.*

3. Fork $=$ (Indep $\setminus$ (Pub $\cup$ Seq)) $\subseteq$ Indep, *and thus the set* Fork *is composed only by sources.*

Lemma 5 $G_1 = (V_1, E_1) \equiv_t G_2 = (V_2, E_2) \ \Rightarrow$ UUDG$(G_1) =$ UUDG(G_2).

Proof 6 *Proof by contradiction. Assume that* UUDG$(G_1) \neq$ UUDG(G_2) *when* $G_1 \equiv_t G_2$. *Then* $\boldsymbol{expr}_{G_1} \neq \boldsymbol{expr}_{G_2}$, *which means that* $\boldsymbol{expr}_{G_1} = (t^1_{\boldsymbol{expr}_{G_1}}, \ldots, t^m_{\boldsymbol{expr}_{G_1}})$, $m \geq 1$, *and* $\boldsymbol{expr}_{G_2} = (t^1_{\boldsymbol{expr}_{G_2}}, \ldots, t^n_{\boldsymbol{expr}_{G_2}})$, $n \geq 1$, *and* $\exists i$ *s.t.* $t^i_{\boldsymbol{expr}_{G_1}} \neq t^i_{\boldsymbol{expr}_{G_2}}$. *Thus, for a particular iteration in* UUDG(G_1), *either* Fork, Seq *or* Join *differs from the respective one in the same iteration of* UUDG(G_2). *If* Dep $= \emptyset$ *then* $V_1 = V_2$ *and* $E_1 = E_2 = \emptyset$. *Thus,* UUDG$(G_1) =$ UUDG(G_2) *and that leads to a contradiction. Otherwise,* $\exists w \in V_2$ *s.t.* $(w, b) \in E_2$ *and* $(a \rightsquigarrow w)_{E_2}$, *and then* $(a \rightsquigarrow w)_{E_1}$ *and* $(w \rightsquigarrow b)_{E_1}$. *Thus, by definition, the content of* Dep *will be the same for both iterations in* UUDG(G_1) *and* UUDG(G_2). *Then, the set* Join *in both iterations must be the same and, furthermore, the set* Seq, *and the set* Fork, *which leads to a contradiction.*

This result guarantees that the UUDG algorithm will exploit the same amount of parallelism with two graphs that are in the same equivalence class $\equiv_t$. Now, the total correctness of the UUDG algorithm will be proved:

Lemma 6 (Partial Correctness of Algorithm 4.6) *Let t, t_1 and t_2 be terms from the grammar presented in Definition 3. Let H_v correspond to the handler associated to a particular parallel goal v. The UUDG algorithm is partially correct with respect to the precondition $\{|V_i| \geq \emptyset;\ |E_i| \geq \emptyset\}$ and the postcondition $\{G_{expr} \equiv_t G_i\}$ where **expr** is the parallelized clause generated by the algorithm, $G_{expr} = (V_{expr}, E_{expr})$ and $G_i = (V_i, E_i)$, such that:*

$$
\begin{aligned}
V_{expr} \quad &= \quad \bigcup_{t \in \texttt{expr}} \texttt{get\_literals}(v) \ \text{ s.t. } \ t = v \vee t = H_v\texttt{<\&!} \vee t = H_v\texttt{<\&} \\
E_{expr} \quad &= \quad \Big(\bigcup_{t_1 \in \texttt{expr}} \{(a,b) \mid \{a,b\} \subseteq \texttt{get\_literals}(v) \ \wedge \ a \prec b\} \Big) \quad \cup \\
&\qquad \Big(\bigcup_{t_2 \in \texttt{expr}} \{(w,v) \mid w = \texttt{last}(\texttt{get\_literals}(x)),\ v \in V_i \setminus P^{t_2}\} \Big) \\
&\qquad \text{s.t. } \ t_1 = v\texttt{\&>}H_v \vee t_1 = v\texttt{\&!>}H_v \\
&\qquad \textit{and } t_2 = x \vee t_2 = H_x\texttt{<\&} \vee t_2 = H_x\texttt{<\&!}
\end{aligned}
$$

where $\texttt{last}(S) = x$ iff $\forall y \in S,\ (y \neq x \ \wedge \ y \prec x)$, i.e., $\texttt{last}(S)$ is a function which returns the rightmost literal in the clause associated to any of the nodes in S.
*and $P^f = \{z \mid \forall y \in$ **expr**$,\ (y = z \vee y = z\texttt{\&>}H_z \vee y = z\texttt{\&!>}H_z),\ y \prec f\}$, i.e., the set of all literals published for parallel execution in **expr** before f.*

Proof 7 *The first union of edges in E_{expr} states that there will be an edge connecting each node that is in the same group. Because of Theorem 1, all nodes in the same group are mutually dependent and thus those nodes will just form an equivalent graph to G_i with respect to $\equiv_t$. Because of Lemma 5, the final parallel expression will be the same.*

Thus, it can be assumed no grouping of nodes needs to be performed, and therefore the postcondition can be simplified to:

$$
\begin{aligned}
V_{expr} &= \{v \mid v \in \texttt{expr},\ H_v\texttt{<\&} \in \texttt{expr},\ H_v\texttt{<\&!} \in \texttt{expr}\} \\
E_{expr} &= (\bigcup_{t \in \texttt{expr}} \{(w,v) \mid v \in V_i \setminus P^t\}) \\
&\quad \text{s.t. } t = w \vee f = H_w\texttt{<\&} \vee f = H_w\texttt{<\&!}
\end{aligned}
$$

Let us prove this now by induction on V*.*

- *Base case: if* $|V| = 0$ *then* $\texttt{expr} = \texttt{true}$*, and thus* $V_{expr} = \emptyset$*,* $E_{expr} = \emptyset$*. Therefore,* $G_{expr} \equiv_t G_i$*.*

- *Induction hypothesis: assuming that the* UUDG *algorithm is started with a particular dependency graph* G *that makes the precondition* true*, the resulting values make the postcondition* true*.*

- *Inductive step: Let the invariant of the loop be* $I = \{G_i \equiv_t (G \cup G_{expr})\}$*. It is necessary to prove that the invariant still holds after executing an iteration of the loop.*

  - *Since* G *is simplified to* $G|_{(V \setminus \text{Join}) \setminus \text{Seq}}$ *at the end of the iteration,* V *will be* $(V \setminus \text{Join}) \setminus \text{Seq}$*. Since* $\texttt{expr}$ *is increased with some nodes to be* Fork*ed,* Seq*uentialized or* Join*ed,* V_{expr} *will be* $V_{expr} \cup \{v \mid (v \in \text{Seq}) \vee (v \in \text{Join})\}$*, because the annotations* v*,* H_v<& *and* H_v<&! *correspond to those for the sets* Seq *and* Join*. Thus,* V_{expr} *will be* $V_{expr} \cup (\text{Join} \cup \text{Seq})$*. Therefore,* $((V \setminus \text{Join}) \setminus \text{Seq}) \cup (V_{expr} \cup (\text{Join} \cup \text{Seq})) = V \cup V_{expr} = V_i$*.*

  - *Since* G *is simplified to* $G|_{(V \setminus \text{Join}) \setminus \text{Seq}}$ *at the end of the iteration,* E *will be* $(E \setminus \{(u,v) \mid u \in \text{Join} \cup \text{Seq},\ v \in V\}) \setminus \{(u,v) \mid (v \in \text{Join} \cup \text{Seq},\ u \in V\})$*. For Lemma 4, the node* u *must be a source and then the new value of* E *is simplified to* $(E \setminus \{(u,v) \mid u \in \text{Join} \cup \text{Seq},\ v \in V\})$*. Moreover, since* v *is a dependent node, it can be only a node that has not been published*

yet, and so E is simplified to:

$$(E \setminus \{(u,v) \mid u \in \text{Join} \cup \text{Seq}, v \in ((V \setminus \text{Pub}) \setminus \text{Fork})\})$$

Since **`expr`** *is increased with some nodes to be* Fork*ed,* Seq*uentialized or* Join*ed, E_{expr} will be:*

$$E_{expr} \cup (\bigcup_{u \in (\text{Seq} \cup \text{Join})} \{(u,v) \mid v \in ((V \setminus \text{Pub}) \setminus \text{Fork})\}) =$$
$$= \{(u,v) \mid u \in \text{Join} \cup \text{Seq},\ v \in ((V \setminus \text{Pub}) \setminus \text{Fork})\}$$

Therefore, $E \cup E_{expr} = E_i$.

In addition, $I \wedge \{V = \emptyset\} \Rightarrow \{G_{expr} \equiv_t G_i\}$, which corresponds to the postcondition of the UUDG algorithm, so it is partially correct.

Lemma 7 (Termination of Algorithm 4.6) *The UUDG algorithm terminates.*

Proof 8 *The set of natural numbers $\mathbb{N}$ is chosen as set with strict well-founded ordering $<$. Let $|V|$ be the termination expression. By definition, $|V| \in \mathbb{N}$ whenever the control of the algorithm starts a new iteration. $|V|$ takes a smaller value in each iteration of the loop with respect to $<$ if either* Join *or* Seq *is not empty. Since* Join $= V \neq \emptyset$ *when* Dep $= \emptyset$ *or else* Join $= I_v \in$ Dep*, which is by definition non-empty, the algorithm terminates.*

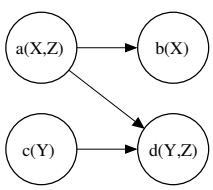
Theorem 2 (Total Correctness of Algorithm 4.6) *The UUDG algorithm is totally correct.*

Proof 9 *Lemma 6 states that the UUDG algorithm is partially correct, and Lemma 7 states that it terminates, so the UUDG algorithm is totally correct.*

In what follows, some examples will show how the UUDG algorithm works.

Example 12 (UUDG Annotation) *This example shows the iterations of the UUDG algorithm in order to parallelize the predicate* `p/3`*, introduced in Section 4.1.1 and whose dependency graph is shown in Figure 4.1(b). Although the first step of the UUDG algorithm groups literals, in the example at hand no grouping can be performed in any iteration. Therefore, literals are taken into account as they appear in the original clause.*

- First iteration*:*



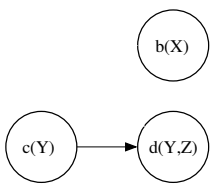
Indep $= \{a, c\}$, Dep $= \{\{a\}, \{a, c\}\}$, SS $= \{a\}$
Seq $= \{a\}$, Fork $= \{c\}$, Join $= \emptyset$
Pub $= \{a, c\}$
Annotation: `p(X,Y,Z) :- c(Y) &> Hc, a(X,Z),`

The first step in the algorithm detects the nodes that are sources *in the graph (literals* `a/2` *and* `c/1`*) and adds them to the set* Indep*. Literals which depend on them and which have only incoming edges from sources are added to the set* Dep*. In order to maximize exploitable parallelism, the least number of literals required to free a dependent literal will be joined. This can be done by choosing the smallest set in* Dep *($\{a\}$, in this case). Thus, literals* `a/2` *and* `c/1` *are to be published for parallel execution and only literal* `a/2` *needs to be joined.*

As an optimization, one of the goals among those which are scheduled for parallel execution and joined in the same iteration can be executed sequentially; this is what happens to literal `a/2`*. After the annotation is done, literals* `a/2`

and `c/1` *are stored in* Pub *and the graph is simplified by removing the node corresponding to* `a/2`*.*

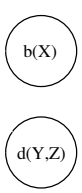
- Second iteration*:*



Indep $= \{b, c\}$, Dep $= \{\{c\}\}$, SS $= \{c\}$
Seq $= \emptyset$, Fork $= \{b\}$, Join $= \{c\}$
Pub $= \{a, b, c\}$
Annotation*:* `b(X) &> Hb, Hc <&,`

Literals `b/1` *and* `c/1` *are sources now, and only* `d/2` *is a dependent node. Literal* `c/1` *was scheduled for parallel execution in the previous iteration and therefore only* `b/1` *needs to be published. Since* `d/2` *needs to be freed,* `c/1` *is waited on. After the annotation is done, literal* `b/1` *is stored in* Pub *and the graph is simplified.*

- Last iteration*:*

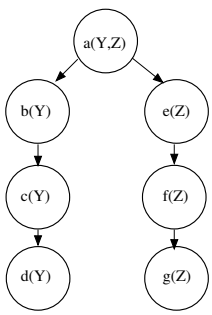


Indep $= \{b, d\}$, Dep $= \emptyset$, SS $= \emptyset$
Seq $= \{d\}$, Fork $= \emptyset$, Join $= \{b\}$
Pub $= \{a, b, c, d\}$
Annotation*:* `d(Y,Z), Hb <&.`

In the last iteration, and as all the nodes are sources, all not-yet-published literals (i.e., `d/2`*) will be scheduled for execution, and the joins of all the remaining literals (i.e.,* `b/1` *and* `d/2`*) will be performed.*

Example 13 (UUDG Annotation with Grouping) *In this example, intuitively, there are two sequences of dependent literals each of which can be independently executed. The second step of the algorithm will need to group both sequences of literals.*

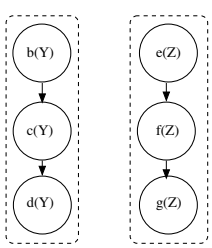
- First iteration*:*



Indep $= \{a\}$, Dep $= \{\{a\}\}$, SS $= \{a\}$
Seq $= \{a\}$, Fork $= \emptyset$, Join $= \emptyset$
Pub $= \{a\}$
Annotation: `a(Y,Z),`

In the first step of the algorithm, literal `a/2` *is scheduled for sequential execution, since the rest of the literals in the clause depend on it. Literal* `a/2` *is then removed from the graph.*

- Second (and last) iteration*:*



Indep $= \{(b,c,d),(e,f,g)\}$, Dep $= \emptyset$, SS $= \emptyset$
Seq $= (e,f,g)$, Fork $= \{(b,c,d)\}$, Join $= \{(b,c,d)\}$
Pub $= \{a,b,c,d,e,f,g\}$
Annotation:
```
(b(Y),c(Y),d(Y)) &> H,
(e(Z),f(Z),g(Z)),
H <&.
```

Literals `b/1`, `c/1` *and* `d/1` *are grouped, and literals* `e/1`, `f/1` *and* `g/1` *are also grouped. The graph is then reduced to one that has only two nodes, which are both sources and which can be executed in parallel. Note that, as in Example 12, all the parallelism is exploited due to the fact that mutually dependent literals are grouped.*

4.2.3 Order-Preserving Annotation: the UOUDG Algorithm

The UOUDG algorithm, presented in Figure 4.9, follows the same idea underlying the UUDG algorithm introduced in Figure 4.6: publish early and join late. However, the UOUDG algorithm has less freedom to publish goals, since the order of solutions is preserved. This is done by respecting the relative order of literals in the original clause, through the use of the relation $\prec$ and the partial function `pred`(L).

As a previous step in each iteration of the algorithm, the function `group_ord`(G, P), shown in Figure 4.10, is called in order to group nodes in a fashion similar to `group_nonord`(G, P) (Figure 4.7). However, for this case the grouping of literals is done in such a way that its order is always preserved.

An important element in the algorithm is *pvt*, the *pivot* vertex, which will be used in order to decide which nodes are to be joined, taking into account that it is not possible to change the order of the solutions. If *Dep* is empty, then all the remaining literals are already independent and can be joined up to the rightmost literal in the clause. Otherwise, the leftmost node among those which have dependencies which can be fulfilled in one step is selected. These dependencies are readily available in *Dep*. Note that as the leftmost node among those which can be joined is selected, joining nodes is delayed as much as possible —or, alternatively, in every step only the joins which are needed to continue one more step are performed. This is aimed

Algorithm: UOUDG(G_i, I_D)
Input: **(1)** A directed acyclic graph $G_i = (V_i, E_i)$.
(2) Determinacy information.
Output: A parallelized clause $\texttt{expr}_{G_i}$ in *unrestricted and* fashion in which the order of the solutions in the original clause is preserved.

begin
  $\texttt{expr}_{G_i} \leftarrow (\texttt{true})$;
  $Pub \leftarrow \emptyset$;
  $G = (V, E) \leftarrow G_i$;
  while $V \neq \emptyset$ **do**
    $G \leftarrow \texttt{group\_ord}(G,\ Pub)$;
    $Indep \leftarrow \{v \mid v \in V,\ \texttt{incoming}(v, E) = \emptyset\}$;
    $Dep \leftarrow \{(v, I_v) \mid v \in V,\ I_v = \texttt{incoming}(v, E),\ I_v \neq \emptyset,\ I_v \subseteq Indep\}$;
    if $Dep = \emptyset$ **then**
      $(pvt,\ Join) \leftarrow (u, V)$ s.t. $\forall(w \in (V \setminus \{u\}))\ .\ w \prec u$;
    else
      $(pvt,\ Join) \leftarrow (u, S)$ s.t. $(u, S) \in Dep \wedge \forall((w, D) \in (Dep \setminus \{(u, S)\}))\ .\ u \prec w$;
    end
    $Seq \leftarrow \{v \mid v \in\ (Indep \setminus Pub),\ (v,pvt)\ \in E,\ v = pred(pvt)\}$;
    $Fork \leftarrow \{v \mid v \in\ (Indep \setminus Pub),\ v \prec pvt\} \setminus Seq$;
    $Join \leftarrow Join \setminus Seq$;
    $Pub \leftarrow Pub \cup (\bigcup_{v \in Fork} \texttt{get\_literals}(v)) \cup \texttt{get\_literals}(\texttt{u})$ s.t. $u \in Seq$;
    $G \leftarrow\ G|_{(V \setminus Join) \setminus Seq}$;
    $\texttt{expr}_{G_i} \leftarrow (\texttt{expr}_{G_i}, \texttt{gen\_body\_ord}(Fork,\ Seq,\ Join,\ I_D))$;
  end
  return $\texttt{expr}_{G_i}$;
end

Figure 4.9: UOUDG annotation algorithm.

at maximizing the number of parallel goals being executed at any moment.

The UOUDG algorithm uses the $\texttt{gen\_body\_ord}(F, S, J, I_D)$ function, which is shown in Figure 4.11, to output a parallelized clause. The function $\texttt{gen\_body\_ord}(F, S, J, I_D)$, as well as the function $\texttt{gen\_body\_nonord}(F, S, J, I_D)$ in Figure 4.8, make use of the auxiliary function $\texttt{get\_literals}(v)$, in addition to some determinism information,

Algorithm: `group_ord`(G, Pub)
Input: **(1)** A nonempty directed acyclic graph $G = (V, E)$.
(2) A set of goals already forked.
Output: A compacted directed acyclic graph $G_f = (V_f, E_f)$ which preserves the order of literals in the grouping.
begin
  forall $v \in V$ s.t. `get_literals`(v) $\not\subseteq Pub$ **and** `incoming`(v, E) $= \emptyset$ **do**
    $NewV \leftarrow \{v\}$;
    $DS \leftarrow \{u \mid u \in V,\ u \notin NewV,\ w \in NewV,\ (w, u) \in E\}$;
    while $DS \neq \emptyset$ **do**
      $NewV' \leftarrow NewV \cup DS$;
      if $(\forall u \in NewV',\ (u = v) \vee (pred(u) \in NewV'))$ **and**
        $(\forall \{v_i, v_j\} \subseteq NewV',\ (v_i \rightsquigarrow v_j)_E \ \vee \ (v_j \rightsquigarrow v_i)_E)$ **and**
        $(\forall e = (v_k, v_l) \in E,\ v_k \notin NewV' \Rightarrow v_l \notin NewV')$ **then**
          $NewV \leftarrow NewV'$;
      else
        break;
      $DS \leftarrow \{u \mid u \in V,\ u \notin NewV,\ w \in NewV,\ (w, u) \in E\}$;
    end
    `set_literals`(v, $NewV$);
    $G \leftarrow G|_{(V \setminus (NewV \setminus \{v\}))}$;
  end
  $G_f \leftarrow G$;
  return G_f;
end

Figure 4.10: Order-preserving grouping of nodes.

by using the auxiliary boolean function `det`(n, I_D), in order to decide whether the optimized versions of the operators `&>/2` (i.e., `&!>/2`) and `<&/1` (i.e., `<&!/1`) are to be used when literals are known to be deterministic.

4.2.4 Correctness Proof of the UOUDG Algorithm

In order to prove the total correctness of the UOUDG algorithm, we will start by proving the total correctness of the `group_ord` function by proving partial correctness

Algorithm: `gen_body_ord`(*Fork*, *Seq*, *Join*, I_D)
Input: **(1)** A set of vertices to be forked.
(2) A set of vertices to be sequentialized.
(3) A set of vertices to be joined.
(4) Determinacy information.
Output: An unrestricted parallelized sequence of literals *Exp*.
begin
Exp ← (`true`);
forall $v_i \in$ *Fork* **do**
if $det(v_i, I_D)$ **then**
Exp ← (*Exp*, `seq(get_literals(v_i)) &!>` H_{v_i});
else
Exp ← (*Exp*, `seq(get_literals(v_i)) &>` H_{v_i});
end
end
if *Seq* = $\{v\}$ **then**
Exp ← (*Exp*, `seq(get_literals(v))`);
end
forall $v_i \in$ *Join* **do**
if $det(v_i, I_D)$ **then**
Exp ← (*Exp*, H_{v_i} `<&!`);
else
Exp ← (*Exp*, H_{v_i} `<&`);
end
end
return *Exp*;
end

Figure 4.11: Order-preserving generation of a parallel body.

and termination.

Lemma 8 (Partial Correctness of Algorithm 4.10) *The function* ***group_nonord*** *is partially correct with respect to the precondition* $\{|V| > \emptyset\}$ *and the postcondition* $\{P_1; P_2; P_3; P_4\}$ *such that:*

1. $P_1 \equiv \bigcup_{v \in V_f}$ ***get_literals***$(v) = V$.

2. $P_2 \equiv \bigcap_{v \in V_f} \texttt{get\_literals}(v) = \emptyset$.

3. $P_3 \equiv$ *no parallelism is lost when constructing* G_f.

4. $P_4 \equiv \forall v \in V_f,\ \forall u \in \texttt{get\_literals}(v),\ ((u = v) \vee (\text{pred}(u) \in \texttt{get\_literals}(v)))$.

Proof 10 *It can be proved in a similar way to Lemma 2. Note that the only difference in the postcondition is the new statement* P_4*, which demands that all nodes to be grouped be consecutive, in order to preserve the order of solutions. That condition will be always* true *because of the first condition of the if-structure.*

Lemma 9 (Termination of Algorithm 4.10) *The function* `group_ord` *terminates.*

Proof 11 *Same proof as in Lemma 3.*

Theorem 3 (Total Correctness of Algorithm 4.10) *The function* `group_ord` *is totally correct.*

Proof 12 *Lemma 8 states that the function* `group_ord` *is partially correct, and Lemma 9 states that it terminates, so the function* `group_ord` *is totally correct.*

The following definition introduces the concept of equivalence class of graphs with respect to a notion of order of literals:

Definition 9 (Equivalence class of graphs w.r.t. $\equiv_o$**)** *Let* $G_1 = (V_1, E_1)$ *and* $G_2 = (V_2, E_2)$ *be two different dependency graphs. Then,* $G_1 \equiv_o G_2 \Leftrightarrow V_1 = V_2 \wedge (\forall v \in V_1,\ ((v, w) \in E_1 \Rightarrow ((v, w) \in E_2 \wedge (\forall x \in V_2 \text{ s.t. } w \prec x,\ (v, x) \in E_2))))$.

Lemma 10 $G_1 = (V_1, E_1) \equiv_o G_2 = (V_2, E_2) \Rightarrow \mathsf{UOUDG}(G_1) = \mathsf{UOUDG}(G_2)$.

Proof 13 *Proof by contradiction. Assume that* UOUDG$(G_1) \neq$ UOUDG(G_2) *when* $G_1 \equiv_o G_2$. *Then* $\mathbf{expr}_{G_1} \neq \mathbf{expr}_{G_2}$, *which means that* $\mathbf{expr}_{G_1} = (t^1_{\mathbf{expr}_{G_1}}, \ldots, t^m_{\mathbf{expr}_{G_1}})$, $m \geq 1$, *and* $\mathbf{expr}_{G_2} = (t^1_{\mathbf{expr}_{G_2}}, \ldots, t^n_{\mathbf{expr}_{G_2}})$, $n \geq 1$, *and* $\exists i$ *s.t.* $t^i_{\mathbf{expr}_{G_1}} \neq t^i_{\mathbf{expr}_{G_2}}$. *Thus, for a particular iteration in* UOUDG(G_1), *either* Fork, Seq *or* Join *differs from the respective one in the same iteration of* UOUDG(G_2). *If* Dep $= \emptyset$ *then* $V_1 = V_2$ *and* $E_1 = E_2 = \emptyset$. *Thus,* UOUDG$(G_1) =$ UOUDG(G_2) *and that leads to a contradiction. Otherwise, pvt and* Join *will be the same in both* UOUDG(G_1) *and* UOUDG(G_2) *because pvt represents the first dependent node with respect to* $\prec$, *and* Join *its dependencies which precede it.* Fork *and* Seq *will also be the same since they are dealing only with predecessors to the pvt, and that leads to a contradiction.*

The total correctness proof of the UOUDG algorithm follows.

Lemma 11 (Partial Correctness of Algorithm 4.9) *The* UOUDG *algorithm is partially correct with respect to the same precondition as in Lemma 6, and postcondition* $\{G_{expr} = (V_{expr}, E_{expr}) \equiv_o (V_i, E_i) = G_i\}$, *where* V_{expr} *and* E_{expr} *have the same value as in Lemma 6.*

Proof 14 *This lemma can be proved in a similar way to Lemma 6. Because of the results of Theorem 3 and Lemma 10, the postcondition can also be simplified. The inductive part of the proof is similar to that of Lemma 6, with the loop invariant* $I = \{G_i \equiv_o (G \cup G_{expr})\}$.

Lemma 12 (Termination of Algorithm 4.9) *The* UOUDG *algorithm terminates.*

Proof 15 *It can be proved in a similar fashion to Lemma 7.*

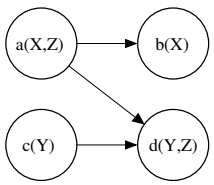
Theorem 4 (Total Correctness of Algorithm 4.9) *The* UOUDG *algorithm is totally correct.*

Proof 16 *Lemma 11 states that the UOUDG algorithm is partially correct, and Lemma 12 states that it terminates, so the UOUDG algorithm is totally correct.*

The following example will sketch how the UOUDG algorithm works.

Example 14 (UOUDG Annotation) *In order to illustrate how the UOUDG algorithm works, in a similar fashion to Example 12, the different iterations in the parallelization process of the predicate `p/3`, whose dependency graph is shown in Figure 4.1(b).*

- First iteration*:*

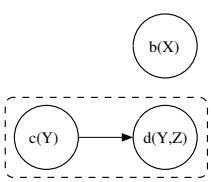


Indep $= \{a, c\}$, Dep $= \{(b, \{a\}), (d, \{a, c\})\}$, pvt $= b$
Seq $= \{a\}$, Fork $= \emptyset$, Join $= \emptyset$
Pub $= \{a\}$
Annotation: `p(X,Y,Z) :- a(X,Z),`

In the first step of the algorithm, both literals `a/2` and `c/1` are both candidates for parallel execution (they are in Indep*). Also, since literals `b/1` and `d/2` only have dependencies with nodes in* Indep*, both literals will be stored in* Dep*, with their respective set of dependencies. Since literal `b/1` is the one in* Dep *with fewer dependencies, it is chosen as pivot, and literal `a/2` marked to be joined in this iteration of the algorithm. Moreover, although literal `c/1` is in* Indep*, only literal `a/2` can be marked to be published for parallel execution, since the order of the literals in the initial clause must be preserved, and literal `b/1` has not been published yet. However, as literal `a/2` is the only one to be published and*

must be joined too, then it is simply selected to be sequentially executed. As a final step, literal `a/2` *is stored in* Pub *and the dependency graph simplified by removing the node corresponding to the literal* `a/2`*. Note how this annotation has less freedom than the* UUDG *annotation in Example 12, always respecting the dependencies that are implicit in the graph.*

- Second (and last) iteration*:*



Indep $= \{b, (c, d)\}$, Dep $= \{\}$, pvt $= (c, d)$
Seq $= \{(c, d)\}$, Fork $= \{b\}$, Join $= \{b\}$
Pub $= \{a, b, c, d\}$
Annotation: `b(X) &> Hb, c(Y), d(Y,Z), Hb <&.`

In this iteration, both literals `b/1` *and* `c/1` *are sources. In this case, literals* `c/1` *and* `d/2` *are compacted into a single node in the graph. Thus, all the nodes in the graph are sources and can be scheduled for parallel execution.*

Once this iteration finishes, the initial clause is unrestrictedly parallelized and the order of the literals in the initial clause, given by the operator `&>/2`*, preserved.*

Finally, note that more parallelism is exploited (in fact, all the possible parallelism) with the UUDG *annotation in Example 12 than with the* UOUDG *annotation, since the order of the literals in the clause does not have to be preserved.*

4.3 Granularity-Aware Annotation

The annotators presented in this chapter are concerned only with capturing independence and mark as available for parallel execution any goals determined to be independent. However, in practice, and even on quite efficient multicore architectures, parallel execution involves overheads associated with task creation, scheduling, locking, memory management, etc.

As a result, if the granularity of parallel tasks, i.e., the "work available" underneath them, is too small, it may happen that the costs of parallel execution are larger than the benefits. This makes it desirable to take task granularity into account in parallel execution. Granularity control has been studied in the context of traditional programming [KL88, MG89], functional programming [Hue93, HLA94], and also logic programming [Kap88, DLH90, ZTD+92, DL93, LGHD96].

It is interesting to devise annotation algorithms which take task granularity into account. In the traditional approach to this problem [DLH90, ZTD+92, DL93, LGHD96] granularity control is applied after the annotation process. While this approach has been shown to indeed bring improved speedups [LGHD96] the following motivational example illustrates its limitations:

Example 15 *Consider the following clause (the arguments of the literals, which create the dependencies between them, are omitted for simplicity):*

```
p :- a, b, c, d, e.
```

Assume that the dependencies detected between the subgoals of `p` *are* `dep(a,b)`, `dep(a,d)`, `dep(b,e)`, `dep(c,d)`, *where we denote by* `dep(X,Y)` *that* `Y` *depends on (and therefore must be executed before)* `X`.

Two of the possible parallelizations for this clause are `( a, b & c, d & e )` *and* `( c & ( a, b, e ), d )`*, both of which respect the dependencies.*

Let $T(\overline{n})$ *be the vector of cost functions for each literal of the clause.* $T(i) < T(j)$ *means that the cost of the subgoal* `i` *is smaller than the cost of* `j`*. Assume that* $T(a) < T(c) < T(e) < T(b) < T(d)$*.*

It can be seen that, in principle, the second parallel expression exploits more parallelism than the first one. However, the first annotation will probably result in better speedups, since literal `d` *(which is large) is executed sequentially in the second annotation and in parallel with* `e` *in the first one. Assume now that only* `b` *and* `d` *are worth being parallelized. In this case, the first annotation is also inefficient, because* `b` *is scheduled to be executed in parallel with* `c`*, of small cost. In this case, a better solution would be* `( a, c, ( b & d ), e )`*. Note that post-processing of the previous annotations to perform granularity control [LGHD96] would not achieve this result.*

This type of annotation requires the use of functions to evaluate whether the cost function of a literal is bigger or smaller than the cost function of another literal. It is indeed sometimes possible to determine precisely at compile time the maximum or minimum among several cost functions. This can also be done heuristically, e.g., comparing only the order information, looking for crossover points between the functions, evaluating the functions on user provided annotations regarding bounds on input data sizes (for example, saying that a given input array is larger than nxn), etc.

In addition, it is necessary to evaluate at compile time if a literal is "sufficiently large". For this case, also different heuristics can be used. Note that the classic annotators are in fact using the simple compile-time heuristic that all tasks are sufficiently

large and of the same cost. When size determinations and comparisons cannot be resolved at compile-time, run-time tests can be introduced in the parallelized clause during the annotation process (as done in [LGHD96], but in that case after the annotation) with the idea of evaluating them at run-time to determine whether a particular parallelization can be performed.

A prototype of a granularity-aware annotation algorithm has been implemented and integrated within the CiaoPP system [HPBLG05].

4.4 Performance Evaluation

The proposed annotation algorithms have been integrated into the Ciao/CiaoPP system [HPBLG05]. Information gathered by the analyzers on variable sharing, groundness, and freeness is used to determine goal independence, using the libraries available in CiaoPP. Determinism is used in the annotators as described previously.

The execution platform used is the shared-memory high-level implementation of the proposed parallelism primitives [CCH08a] developed as an extension of the Ciao system [BCC+06] as part of this thesis, and presented in Chapter 5. This implementation is an evolution of [Her86, HG91] based on raising the level of certain components to the source language and keeping only some selected operations (related to thread handling, locking, etc.) at a lower level. This approach does not eliminate altogether modifications to the abstract machine, but it greatly simplifies them.

The impact of the different annotations on the execution time has been evaluated by running a series of benchmarks (briefly described in Table 4.1) in parallel. Table 4.2 shows the speedups obtained *with respect to the sequential execution* (i.e.,

AIAKL	An abstract interpreter for the AKL language.
FFT	An implementation of the Fast Fourier transform.
FibFun	A version of the Fibonacci program written in functional notation.
Hamming	A program to compute the first N Hamming numbers.
Hanoi	A program to compute the movements to solve the well-known puzzle, as proposed in [MBdlBH99].
Takeuchi	Computes the Takeuchi function.
WMS2	A scheduler assigning a number of workers to a series of jobs.

Table 4.1: Benchmark programs

they are *actual* speedups, which is the reason why some speedups start below 1 for, e.g., one thread), when using from 1 to 8 threads.

The machine used is a Sun UltraSparc T2000 (a *Niagara*) with 8 4-thread cores. In the performance results shown in Table 4.2, not more than 8 cores are used since in that case, and due to access to shared units, speedups are sublinear even for completely independent tasks.

The *fork-join* annotators chosen to compare with are MEL [MBdlBH99] (which preserves goal order and tries to maximize the length of the parallel expressions) and UDG [Cab04] (which can reorder goals). The MEL algorithm tries to find longest parallel expression by proceeding backwards from the last literal in order to find a hard dependency between two literals and hence finding a point where the expression can be split into two different ones. The UDG algorithm assumes that all the dependence conditions that cannot be completely determined statically are false, in order to eliminate the overhead of evaluating run-time checks in the parallel expression.

However, the limitation (and complication) of these fork-join annotation algorithms comes from the use of the fork-join operator, illustrated in Section 4.1. MEL can add runtime checks to decide dynamically whether to execute or not in parallel.

Benchmark	**Annotator**	**Number of threads**							
		1	2	3	4	5	6	7	8
AIAKL	UMEL	0.97	0.97	0.98	0.98	0.98	0.98	0.98	0.98
	UOUDG	0.97	1.55	1.48	1.49	1.49	1.49	1.49	1.49
	UDG	0.97	1.77	1.66	1.67	1.67	1.67	1.67	1.67
	UUDG	0.97	1.77	1.66	1.67	1.67	1.67	1.67	1.67
FFT	UMEL	0.98	1.76	2.14	2.71	2.82	2.99	3.08	3.37
	UOUDG	0.98	1.76	2.14	2.71	2.82	2.99	3.08	3.37
	UDG	0.98	1.76	2.14	2.71	2.82	2.99	3.08	3.37
	UUDG	0.98	1.82	2.31	3.01	3.12	3.26	3.39	3.63
FibFun	UMEL	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	UOUDG	0.99	1.95	2.89	3.84	4.78	5.71	6.63	7.57
	UDG	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	UUDG	0.99	1.95	2.89	3.84	4.78	5.71	6.63	7.57
Hamming	UMEL	0.93	1.13	1.52	1.52	1.52	1.52	1.52	1.52
	UOUDG	0.93	1.15	1.64	1.64	1.64	1.64	1.64	1.64
	UDG	0.93	1.13	1.52	1.52	1.52	1.52	1.52	1.52
	UUDG	0.93	1.15	1.64	1.64	1.64	1.64	1.64	1.64
Hanoi	UMEL	0.89	0.98	0.98	0.97	0.97	0.98	0.98	0.99
	UOUDG	0.89	1.70	2.39	2.81	3.20	3.69	4.00	4.19
	UDG	0.89	1.72	2.43	3.32	3.77	4.17	4.41	4.67
	UUDG	0.89	1.72	2.43	3.32	3.77	4.17	4.41	4.67
Takeuchi	UMEL	0.88	1.61	2.16	2.62	2.63	2.63	2.63	2.63
	UOUDG	0.88	1.62	2.17	2.64	2.67	2.67	2.67	2.67
	UDG	0.88	1.61	2.16	2.62	2.63	2.63	2.63	2.63
	UUDG	0.88	1.62	2.39	3.33	4.04	4.47	5.19	5.72
WMS2	UMEL	0.85	0.81	0.81	0.81	0.81	0.81	0.81	0.81
	UOUDG	0.99	1.09	1.09	1.09	1.09	1.09	1.09	1.09
	UDG	0.99	1.01	1.01	1.01	1.01	1.01	1.01	1.01
	UUDG	0.99	1.10	1.10	1.10	1.10	1.10	1.10	1.10

Table 4.2: Speedups for several benchmarks and annotators.

In order to make the annotation unconditional (as the annotators presented in this chapter), the conditional parallelism was removed in the places where it was not being exploited. This is why it appears in Table 4.2 under the name *UMEL*.

All the benchmarks executed were parallelized automatically by CiaoPP, starting

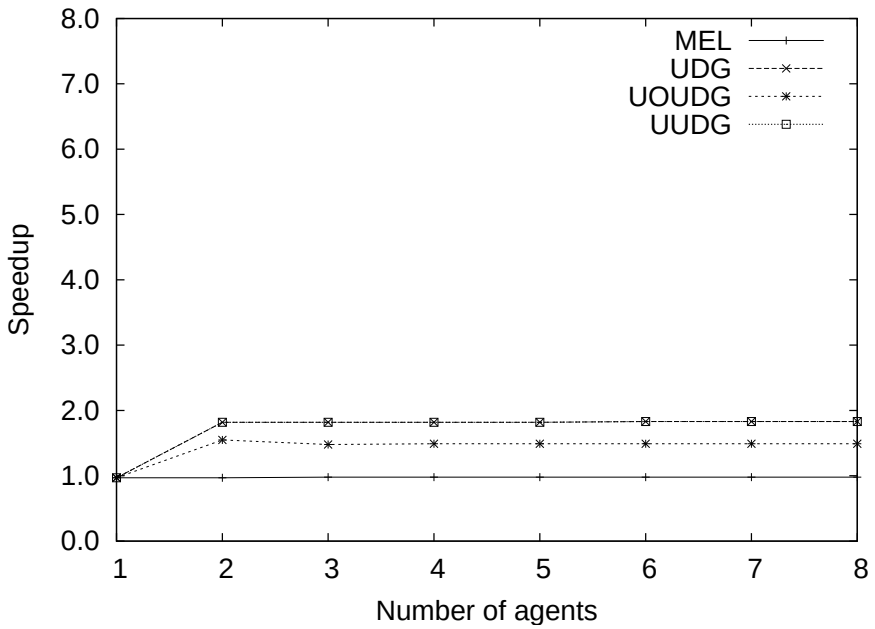


Figure 4.12: Speedups obtained with different annotations for *AIAKL*.

from their sequential code. Since UOUDG and UUDG can improve the results of fork-join annotators only when the code to parallelize has at least a certain level of complexity, not all benchmarks with (independent) parallelism can benefit from using the unrestricted operators. Additionally, comparing speedups obtained with programs parallelized using order-preserving and non-order-preserving annotators is not really meaningful.

Note that the speedups themselves are not the interesting issue of the performance results. Although of utmost practical interest, raw speed is very connected with the implementation of the underlying parallel abstract machine, and improvements on it can be expected to uniformly affect all parallelized programs. Rather, the main focus of attention is in the *comparison* among the speedups obtained using different annotators.

A first examination of the experimental results in Table 4.2, and also in Fig-

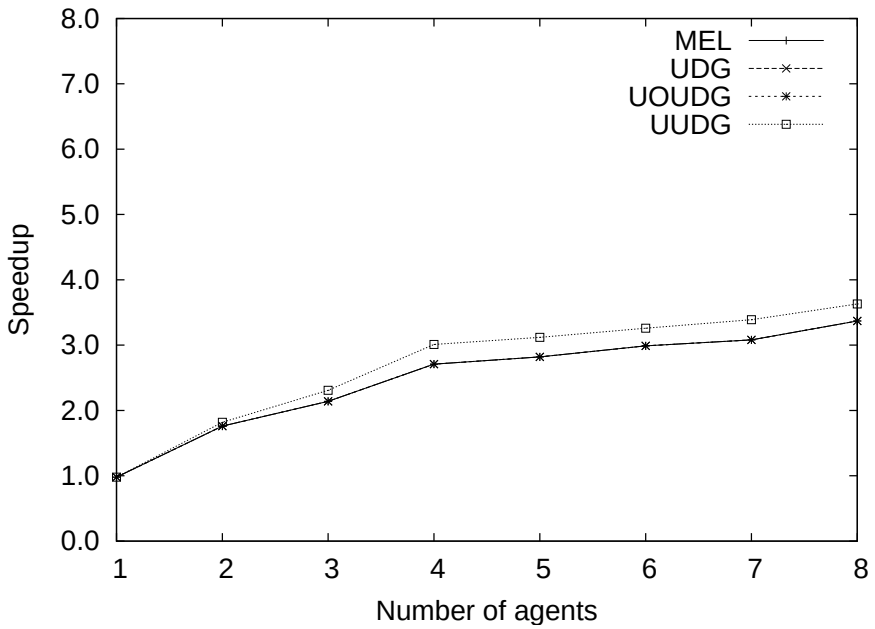


Figure 4.13: Speedups obtained with different annotations for *FFT*.

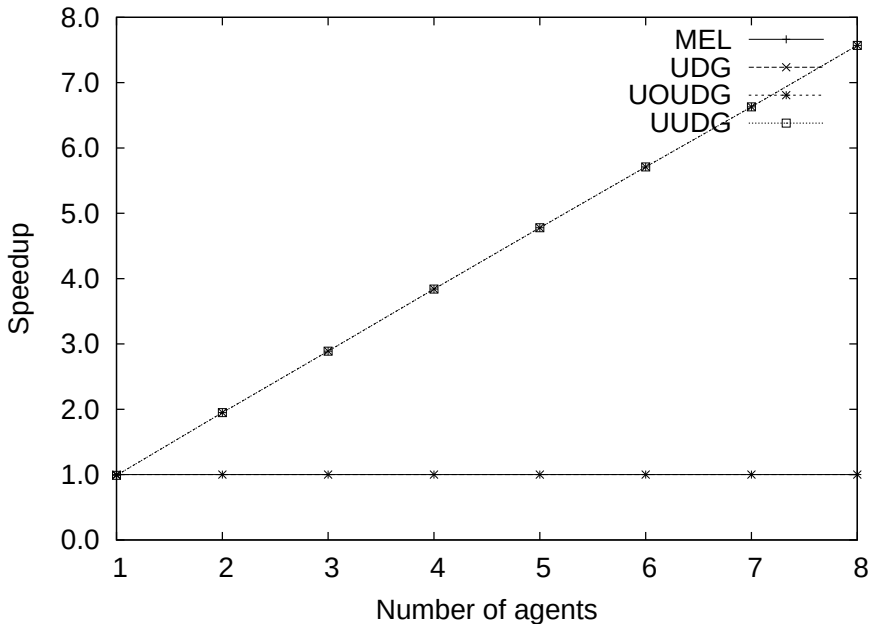


Figure 4.14: Speedups obtained with different annotations for *FibFun*.

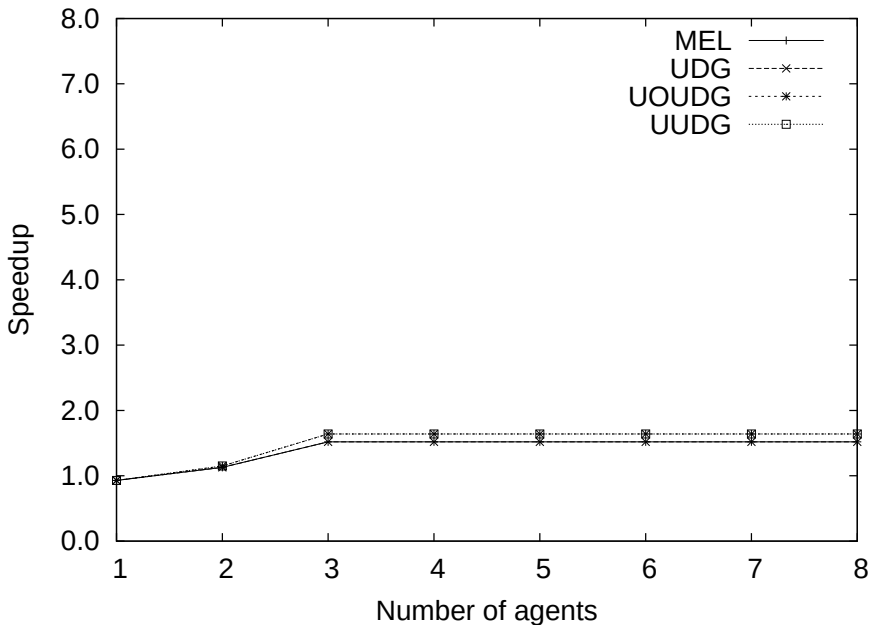


Figure 4.15: Speedups obtained with different annotations for *Hamming*.

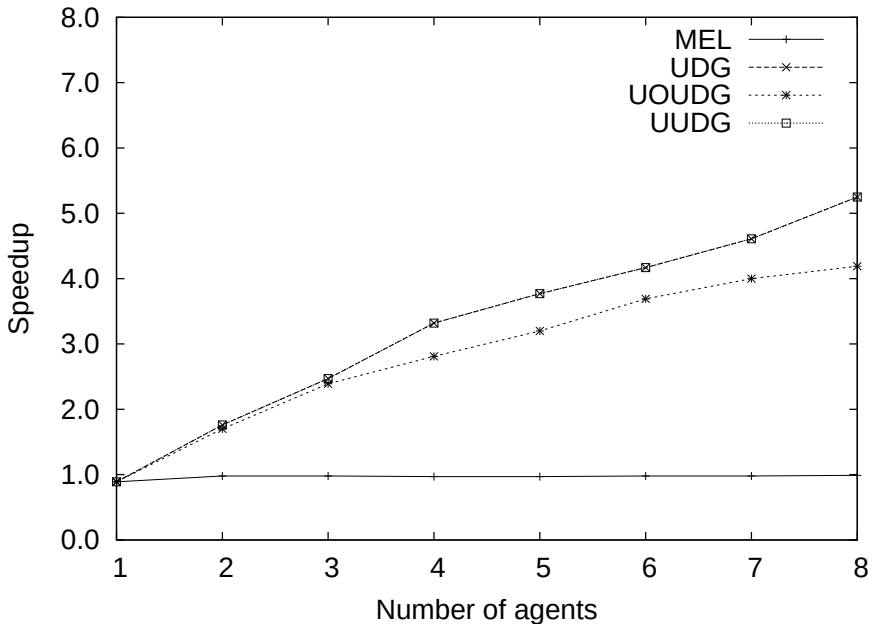


Figure 4.16: Speedups obtained with different annotations for *Hanoi*.

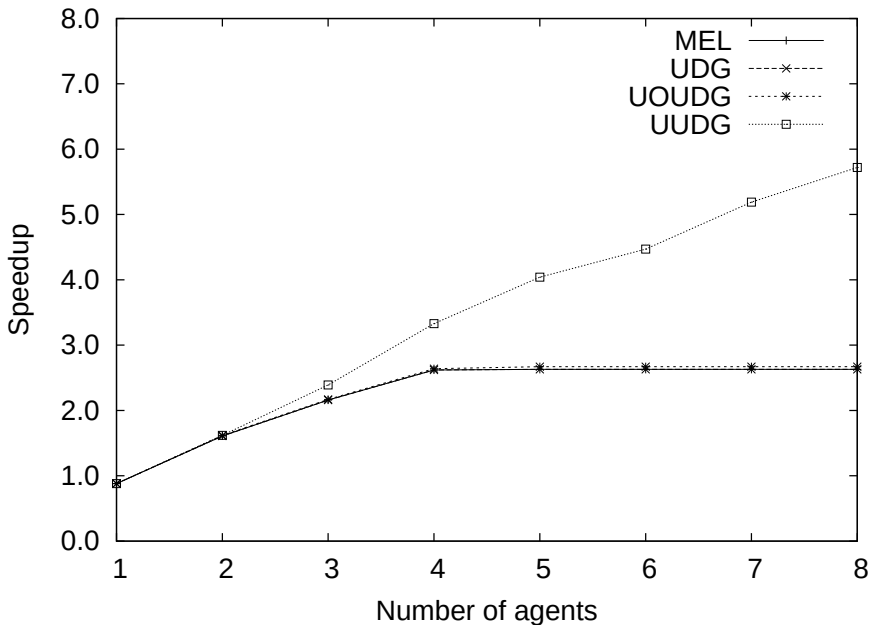


Figure 4.17: Speedups obtained with different annotations for *Takeuchi*.

ures 4.12, 4.13, 4.14, 4.15, 4.16 and 4.17, allows observing that in no case UUDG is worse than any other annotator, and in no case is UOUDG worse than (U)MEL. The results therefore suggest that these should be the annotators of choice (at least compared to the other annotators in CiaoPP). Besides, there are cases where UOUDG is better than UDG, and the other way around, which is in accordance with the non-comparable nature of these two algorithms.

Among the cases in which a better speedup is obtained by some of the U(O)UDG annotators, improvements range between "no improvement" (because no benefit is obtained for some particular cases and combinations of annotators) to an increase of 757% in speedup, with several other levels in between. Also, it is worth pointing out that the difference in speedup does not stabilize in any benchmark (at least in a sizable amount) as the number of threads increases; moreover, in some cases the difference in speedup between the restricted and the unrestricted versions grows sub-

stantially with the number of threads. This can (clearly) be seen in, e.g., Figure 4.17.

Finally, the performance results of three particular benchmarks can be specially highlighted. **FibFun** is the result of parallelizing a definition of the Fibonacci numbers written using the functional notation capabilities of Ciao [CCH06]. Because of the particular order in which code is generated in the (automatic) translation into Prolog (which is somewhat different to the traditional, hand-coded Prolog version), the result is only parallelizable by UOUDG and UUDG, hence the speedup obtained in this case. The case of **Hanoi** is also interesting, as it is the first example in [MBdlBH99]: in the arena of order-preserving parallelizers, UOUDG can extract more parallelism than MEL for this benchmark. Lastly, the **Takeuchi** benchmark has a relatively small loop which only allows parallelizing with a simple `&/2`. However, by unrolling one iteration in the loop the resulting body has dependencies which are complex enough to take advantage of the increased flexibility of the proposed annotators.

4.5 Summary

New annotation algorithms have been proposed which perform a source-to-source transformation of a logic program into an unrestricted independent and-parallel version of itself. Both algorithms rely on the use of more basic high-level primitives than the fork-join operator, and differ on whether the order of the solutions in the original program must be preserved or not. The proposed algorithms have been proved to be correct, and implemented and evaluated in CiaoPP, showing to be more efficient than previous fork-join annotation algorithms.

Chapter 5

High-Level Implementation of Unrestricted IAP

This chapter presents an alternative for implementing and-parallelism in logic programming that raises the core parts of the implementation to the source language level, with the only help of a small number of concurrency-related primitives in order to deal with low-level tasks. This alternative simplifies the complex machinery required by previous solutions, which is in fact difficult to maintain and extend. In addition, it is able to exploit both restricted and unrestricted independent and-parallelism.

First, a simplified version of the implementation that is optimized for the case of non-failing deterministic benchmarks is introduced, and then the complete version of the system for executing non-deterministic independent and-parallel benchmarks is presented. Finally, this chapter will show that the performance results obtained are quite acceptable.

5.1 Shared Memory Implementation for Non-Failing Deterministic IAP

This section presents the solution proposed for the parallel execution of non-failing deterministic parallel programs. Section 5.2 will cover the more complex case of non-deterministic parallel programs.

The proposed implementation divides responsibilities among several layers. User-level parallelism and concurrency primitives intended for the programmer and parallelizers are at the top and written in Prolog. Below, goal publishing, searching for available goals, and goal scheduling are written at the Prolog level, relying on some low-level support primitives for, e.g., locking, low-level goal management or stack set management, with a Prolog interface, but written in C.

In this implementation for shared-memory multiprocessors, and in a similar way to [HG91], agents wait for work to be available, and execute it if so. Every agent is created as a thread attached to an extended WAM stack set. Sequential execution proceeds as usual, and coordination with the rest of the agents is performed by means of shared data structures.

Similarly to the &-Prolog model, agents make new work available to other agents (and also to itself) through a *goal list* which is associated with every stack set and which can be consulted by all the agents. This is an instance of the general class of *work-stealing* scheduling algorithms, which date back at least as far as Burton and Sleep's [BS81] research on parallel execution of functional programs and Halstead's [Hal85] implementation of Multilisp, for functional programs, and the original &-Prolog abstract machine for logic programs [Her86].

In the following sections, the library with the deterministic low-level parallelism

primitives will be introduced, as well as the design and the actual code of the main source-level algorithms used to execute non-failing deterministic goals in parallel.

5.1.1 Low-Level Parallelism Primitives

The low-level layer has been implemented as a Ciao library [BCC+06] ("*apll*") written in C, which provides basic mechanisms to start threads, wait for their completion, publish goals for parallel execution, search for goals, access to O.S. locks, etc. Most of these primitives need to refer to an explicit goal and need to use some information related to its state (whether it has been taken, finished, etc.). Hence the need to pass them a `Handler` data structure which abstracts information related to the goal at hand.

The current list of primitives follows. Note that this is not intended to be a general-purpose concurrency library (such as those available in Ciao and other Prolog systems —in fact, very little of what should appear in such a generic library is here), but simply a list of primitives suitable for efficiently implementing at a higher-level different approaches to exploiting independent and-parallelism. For clarity, the library qualification is added explicitly.

`apll:add_goal(+Goal,+Det,-Handler)` atomically creates a unique *handler* (an opaque structure) associated to `Goal` and publishes `Goal` for any agent to pick it up. `Handler` will henceforth be used in any operation related to `Goal`. `Det` describes whether `Goal` is deterministic or not.

`apll:find_goal(-Handler)` searches for a goal published for parallel execution. If one exists, `Handler` is unified with a handler for it; the call fails otherwise, and it will succeed at most once per call.[1]

[1]Different versions exist of this primitive which can be used while implementing different

`apll:goal_available(+Handler)` succeeds if the goal associated to `Handler` has not been picked up yet, and fails otherwise.

`apll:retrieve_goal(+Handler,-Goal)` unifies `Goal` and the goal initially associated to `Handler`.

`apll:goal_finished(+Handler)` succeeds if the execution state of the goal associated to `Handler` is finished, and fails otherwise.

`apll:set_goal_finished(+Handler)` sets to finished the execution state of the goal associated to `Handler`.

`apll:waiting(+Handler)` succeeds when the execution state of the agent which published the goal associated `Handler` is suspended and fails otherwise.

Additionally, a set of locking primitives is provided to synchronize thread executions and to obtain mutual exclusion at the Prolog level. Agents are synchronized by using two different locks: one which is used to ensure mutual exclusion when dealing with shared data structures (i.e., when adding new goals to the list), and another one which is used to synchronize the agent waking up when `<&/1` is waiting for either more work to be available, or the execution of a goal picked up by some other agent to finish. Both can be accessed with specific (`*_self`) predicates to specify the ones belonging to the calling agent. Otherwise, they are accessed through a goal `Handler`, and then the locks accessed are those belonging to the agent which created the goal that `Handler` refers to (i.e., its *creator*).

`apll:suspend` suspends the execution of the calling thread.

`apll:release(+Handler)` releases the agent which created `Handler` (which could have suspended itself with the above described predicate).

goal scheduling strategies.

`apll:release_some_suspended_thread` selects one out of any suspended threads and resumes its execution.

`apll:enter_mutex(+Handler)` attempts to enter a mutual exclusion by using the lock of the agent associated to `Handler`, in order to access its shared variables.

`apll:enter_mutex_self` same as above, with the agent's own mutex.

`apll:exit_mutex(+Handler)` *signals* the lock in the realm of the agent associated to `Handler` in order to exit mutual exclusion.

`apll:exit_mutex_self` same as above with the calling thread.

The next section will clarify how these primitives are intended to be used.

5.1.2 Implementation

Based on the primitives presented in the previous section, the algorithms for the user-level primitives will be now developed.

High-level Goal Publishing:

In the execution model proposed, a particular strategy will be implemented in which rather than, e.g., having idle agents busily looking for work, such agents are suspended and resumed in a more organized way depending on availability of work (this strategy is also the one used in the experiments).

A call to `&!>/2` implies publishing the goal for parallel execution, i.e., making it available for other agents to pick up for execution. Figure 5.1 shows the (simplified)

```
Goal &!> Handler :-
   apll:add_goal(Goal,det,Handler),
   apll:release_some_suspended_thread.
```

Figure 5.1: Source code for publishing a deterministic parallel goal.

```
Handler <&! :-
    apll:enter_mutex_self,
    (
        apll:goal_available(Handler) ->
        apll:retrieve_goal(Handler,Goal),
        apll:exit_mutex_self,
        call(Goal)
    ;
        apll:exit_mutex_self,
        perform_other_work(Handler)
    ).
```

Figure 5.2: Source code for performing a deterministic goal join with continuation.

Prolog code implementing this functionality. The code shown can be expanded in line but it is shown as a meta-call for clarity.

First, the goal is published for parallel execution. Second, the current thread will signal any suspended agents that there is new work available. As will be seen later, the agent receiving the signal will resume its execution, pick up the new parallel goal, and start its execution.

After executing `Goal &!> H`, the *handler* `H` will hold the state of `Goal`, which can be inspected both by the thread which publishes `Goal` and by any thread which picks up `Goal` to execute it. Therefore, in some sense, `H` takes the role of the *parcall frame* in [HG91], but it is not placed in the environment —it goes to the heap instead, as in [PGH95]. Threads can communicate and synchronize through the handler in

```
perform_other_work(Handler) :-
    apll:enter_mutex_self,
    (
        apll:goal_finished(Handler) ->
        apll:exit_mutex_self
    ;
        (
            find_goal_and_execute ->
            true
        ;
            apll:exit_mutex_self,
            apll:suspend
        ),
        perform_other_work(Handler)
    ).
```

Figure 5.3: Source code for performing some other work when available.

order to consult and update the state of `Goal`. This is especially important when executing `H <&!`.

Performing Goal Joins:

Figures 5.2 and 5.3 provide code implementing `<&!/1` (the deterministic version of `<&/1`).

First, the thread needs to check whether the goal has been picked up by some other thread, using `apll:goal_available/1`. In order to avoid concurrency problems, the thread creates a mutual exclusion using its own lock, since it is accessing its own data structures. If the goal has not been picked up yet by another agent then the publishing agent exits the mutual exclusion and executes it locally, and `<&!/1` will succeed trivially.

If the goal has been picked up by another agent and its execution has finished

```
find_goal_and_execute :-
    apll:find_goal(Handler),
    apll:exit_mutex_self,
    apll:retrieve_goal(Handler,Goal),
    call(Goal),
    apll:enter_mutex(Handler),
    apll:set_goal_finished(Handler),
    (
        apll:waiting(Handler) ->
        apll:release(Handler)
    ;
        true
    ),
    apll:exit_mutex(Handler).
```

Figure 5.4: Source code for finding a parallel goal and executing it.

then `<&!/1` will automatically succeed. Note that a mutual exclusion is entered again in order to safely check the status of the execution. In that case, the bindings made during goal execution are, naturally, available, since the implementation is a shared-memory one. If the goal execution has not finished yet then the thread will not suspend right away. Instead, it will search for more work in order to keep itself busy, and it will only suspend if there is definitely no work to perform at the moment. This ensures that overall efficiency is kept at a reasonable level, as will be seen in Section 5.3. Note that all this process is protected from races when accessing shared variables by using locks for mutual exclusion and conditional synchronization.

Figure 5.4 presents the source code which searches for a goal available and executes it. `find_goal_and_execute/0` will fail if there is no goal available. If one is found then the thread will pick it up, execute it, create a mutual exclusion with the lock of the publishing agent (since the handler associated to the goal resides in the publishing agent) in order to mark the execution as finished and resume the execution of the publishing agent, if suspended.

```
create_agents(0) :- !.
create_agents(N) :-
    N > 0,
    conc:start_thread(agent),
    N1 is N - 1,
    create_agents(N1).
```

```
agent :-
    apll:enter_mutex_self,
    (
        find_goal_and_execute ->
        true
    ;
        apll:exit_mutex_self,
        apll:suspend
    ),
    agent.
```

Figure 5.5: Source code for creating parallel agents.

In that case, the publishing agent (suspended in `eng_suspend/0`) will check which situation applies after resumption and act accordingly after recursively invoking the predicate `perform_other_work/1`.

Agent Creation:

Agents are generated using the `create_agents/1` predicate which launches a number of O.S. threads using the `start_thread/0` predicate imported from a generic concurrency library (thus the `conc` prefix used, again, for clarity). Every one of these threads executes continuously the `agent/0` code which takes care of searching for more work or suspending, if that is the case (Figure 5.5). Thus, during normal execution agents are either sleeping because there is nothing to execute or working on some goal. It is assumed for simplicity that agent creation is in general performed at system startup or just before starting a parallel execution.

Higher-level predicates are however provided in order to manage threads in a more flexible way. For instance, `ensure_agents/1` makes sure that a given number of executing agents is available. In fact, agents can be created lazily, and added or deleted dynamically as needed, depending on machine load. However, this interesting

issue of thread throttling is beyond the scope of this thesis.

5.2 Shared Memory Implementation for Non-Deterministic IAP

Section 5.1 proposed a high-level implementation that raised some of the main components of the implementation to the source level, and was able to exploit the flexibility provided by unrestricted and-parallelism, i.e., not limited to fork-join operations.

However, that solution provided a solution which is only valid for the parallel execution of goals which have exactly one solution each, thus avoiding some of the hardest implementation problems classically found in and-parallel execution. While it can be argued that a large part of application execution is indeed single-solution, on one hand this cannot always be determined a priori, and on the other there are also cases of parallelism among non-deterministic goals, and thus a system must offer a complete implementation, capable of coping with parallel non-deterministic goals, in order to be realistic.

In this section, a high-level implementation will be presented, which is able to exploit unrestricted IAP over *non-deterministic* parallel goals, while maintaining the optimizations of previous solutions for non-failing deterministic parallel goals. This proposal provides solutions for the trapped-goal and garbage-slot problems, and is able to cancel the execution of a parallel goal when needed.

Although the execution model is based on the *multi-sequential, marker model*, there exist significant differences between this proposal and the &-Prolog run-time model, which will be presented in the following sections.

5.2.1 Goal Stacks vs. Goal Lists

In this execution model, each agent is extended with a goal list, the functionality of which is similar to that of the goal stack in the &-Prolog run-time model. The goal list entries store pointers to those goals which have been prepared for parallel execution, and thus agents which are idle can search for parallel goals to execute by consulting the goal lists of the rest of the agents. Goal lists are accessed atomically so as to avoid races when updating them.

A list is used instead of the traditional stack due to the greater flexibility needed in order to deal with the *unrestricted* nature of the `&>/2` and `<&/1` operators (instead of, or in addition to `&/2`): goals can be joined in any order —not necessarily the inverse to the order in which they were published— and, in the case of goal cancellation, arbitrary goal entries inside the list may have to be removed.

For instance, the conjunction $(g_1\texttt{\&}g_2\texttt{\&}\ldots\texttt{\&}g_n)$ can be executed as:

$$(g_1\texttt{\&>}H_1, g_2\texttt{\&>}H_2, \ldots, g_n, \ldots, H_2\texttt{<\&}, H_1\texttt{<\&})$$

as per Equation (2.1), but in fact any order for the goal joins would be equally correct.

Goal lists are implemented in C as doubly-linked lists. This makes it possible to experiment with different scheduling strategies and gives more flexibility for the case of unrestricted and-parallel execution.

5.2.2 Parcall Frames vs. Handlers

As mentioned in Section 2.5, parcall frames in the &-Prolog run-time model are additional (environment) stack frames used for the coordination and synchronization

of the parallel execution. In &-Prolog a parcall frame is created as soon as a parallel call is made, and it has a slot for each of the literals $g_1 , g_2 \ldots g_n$ in the parallel call $g_1 \& g_2 \& \ldots \& g_n$, in order to keep track of the execution of each of these goals.

In most WAM implementations the handling of environments is relatively brittle and introducing different elements in the environment stack complicates things. As an alternative to parcall stack frames, the proposal makes use of *handlers*, heap structures accessible from source-level code, introduced in Chapter 2. Each handler is associated to a particular parallel goal and used for synchronization between the publishing agent and the agent which picks up the parallel goal. In this implementation, handlers will store information such as, e.g., a pointer to the actual parallel goal, a pointer to its location in the goal list (to remove it from there in case the goal is not taken by any other agent), a field to mark the goal as deterministic or not, the state of the goal execution, and pointers to both the publishing agent and the executing agent in order to release their execution when so needed.

5.2.3 Markers vs. (Prolog) Choice Points

As introduced in Section 2.5, *markers* are used in the &-Prolog run-time model to set boundaries between different sections in the stack, each of them corresponding to the *segment* of execution of a parallel goal. This separation of segments in the stack is used to provide a solution to the *trapped goal* problem [HN86]. Markers are also used in &-Prolog to implement storage recovery mechanisms during backtracking of parallel goals, in order to solve the *garbage slot* problem [HN86].

The proposal to avoid the use of new stack frames to implement markers is the creation instead of normal choice points, and to do so in a very simple way by creating alternatives (through predicates with more than one clause) directly in the source-

level (Prolog) code of the scheduler (see Section 5.2.4). This is done whenever a parallel goal is to be executed, as shown in Figure 5.6(e). In addition to that, pointers to the choice points that mark the beginning and the end of the goal execution will be stored in the handler associated to that goal, in order to delimit the segment of execution in such a way that those limits can be accessed during backwards execution. This is also done in part at the source level. Section 5.2.4 provides further explanation of how backwards execution over parallel goals is performed using these normal choice points.

5.2.4 Implementation

Figure 5.6 presents a sketch of the high-level implementation of the scheduler for unrestricted IAP. The library qualification of the primitives is omitted for clarity. The implementation is an extension of that presented in Section 5.1. Similarly to it, agents are created with a small stack (which can grow on demand) and they wait for some work to be available. They do not continuously search for new tasks to be performed, in order to avoid active waiting.[2]

First, when an agent is created, in a fashion very similar to the behavior of the agents presented in Section 5.1, it executes the code shown in Figure 5.6(f), and during normal execution it will start working on the execution of some goal, or will sleep because there is no task to perform (yet). An agent searches for new parallel goals to execute by using a work-stealing scheduling algorithm based on those in [Her86, HG91].

Figure 5.6(a) presents the code for the `&>/2` primitive, which publishes a goal for parallel execution. A pointer to the parallel goal is added to the goal list of the

[2]This decision was taken because it gave slightly better speedups in the experiments and it is in general good usage of a multiuser system.

```
Goal &> Handler :-
   add_goal(Goal,nondet,Handler),
   undo(cancellation(Handler)),
   release_some_suspended_thread.
```

(a) Non-deterministic goal publishing.

```
Handler <& :-
   enter_mutex_self,
   (
      goal_available(Handler) ->
      exit_mutex_self,
      retrieve_goal(Handler,Goal),
      call(Goal)
   ;
      check_if_finished_or_failed(Handler)
   ).
Handler <& :-
   add_goal(Handler),
   release_some_suspended_thread,
   fail.
```

(b) Goal join and speculation.

```
check_if_finished_or_failed(Handler) :-
   (
      goal_finished(Handler) ->
      exit_mutex_self,
      sending_event(Handler)
   ;
      (
         goal_failed(Handler) ->
         exit_mutex_self,
         fail
      ;
         suspend,
         check_if_finished_or_failed(Handler)
      )
   ).
```

(c) Checking status of goal execution.

```
sending_event(_).
sending_event(Handler) :-
   enter_mutex_self,
   enter_mutex_remote(Handler),
   set_goal_tobacktrack(Handler),
   add_event(Handler),
   release_remote(Handler),
   exit_mutex_remote(Handler),
   check_if_finished_or_failed(Handler).
```

(d) Sending event to executing agent.

```
call_handler(Handler) :-
   retrieve_goal(Handler,Goal),
   save_init_execution(Handler),
   call(Goal),
   save_end_execution(Handler),
   enter_mutex(Handler),
   set_goal_finished(Handler),
   release(Handler),
   exit_mutex(Handler).
call_handler(Handler) :-
   enter_mutex(Handler),
   set_goal_failed(Handler),
   release(Handler),
   metacut_garbage_slots(Handler),
   exit_mutex(Handler),
   fail.
```

(e) High-level markers definition.

```
agent :-
   enter_mutex_self,
   work,
   agent.
agent :- agent.

work :-
   (
      read_event(Handler) ->
      (
         more_solutions(Handler) ->
         move_execution_top(Handler)
      ;
         move_pointers_down(Handler)
      ),
      exit_mutex_self,
      fail
   ;
      (
         find_goal(H) ->
         exit_mutex_self,
         call_handler(H)
      ;
         exit_mutex_self,
         suspend,
         work
      )
   ).
```

(f) Agent code.

Figure 5.6: High-level solution for unrestricted IAP.

agent, and a signal is sent to one of the agents that are currently waiting for some task to do. This agent will resume its execution, pick up the goal, and execute it. The communication and synchronization between both agents will be performed via the handler created for that goal. In addition, when the `&>/2` primitive is reached in backwards execution, the memory reserved by the handler is released. Also, if the goal was taken by another agent and the goal execution was not finished yet, then a signal is sent to the executing agent to cancel its execution. This is done with the `cancellation/1` primitive. This operation for cancellation avoids performing unnecessary work and increases the overall performance of the system, as it will be shown in Section 5.3. Moreover, in order to be able to execute this operation in the presence of cuts in the code of the clause, it is invoked via the `undo/1` predicate.

Figure 5.6(b) presents the implementation of the `<&/1` operator. First, the publishing agent needs to check whether the goal was picked up by some other agent or not. If it was not taken then the publishing agent will remove it from the goal list and execute it locally (using `call/1`), and then it will continue executing scheduler code. If the goal was taken by some other agent then its status will be checked (i.e., to know whether the goal execution has already finished or failed) as shown in Figure 5.6(c). If the goal execution fails then the parallel goal will be added to the goal list of the publishing agent, so it can be taken and reexecuted by some other agent. This is a form of speculative execution, since the reexecution of that literal may not be needed for the actual computation. However, it increases the actual parallelism in the system, and note that the goal execution would be cancelled if the corresponding `&>/2` was reached in backtracking.

If the goal execution succeeds and `<&/1` is reached on backtracking, then backwards execution needs to be performed over the parallel goal. If the goal was not taken by some other agent then backwards execution is trivially performed. If it

was picked up by some other agent then the publishing agent sends a signal to the executing agent with a request for a new solution for that goal. The executing agent will serve the signal as soon as it is able. In order to enable this communication, each agent has an *event queue* from which the agent reads and removes events consisting of pointers to handlers associated to the goals to be backtracked over. The primitives which perform this communication are `add_event/1`, which inserts a new pointer to a handler in the event queue of the agent which executed the associated goal, and `read_event/1`, which either reads and deletes an item from the event queue in order to perform backwards execution over the parallel goal associated to it, or fails if the event queue is empty. These primitives extend the list of primitives shown in Section 5.1.1. Figure 5.6(d) presents the source code which is executed to add the corresponding event to the executing agent, releasing its execution if it was suspended.

When an agent reads an event, as shown in Figure 5.6(f), backwards execution over a parallel goal needs to be performed. If the segment of execution is at the top of its stack, then the agent will invoke `fail/0` and a new solution will be obtained. However, it might be the case that the segment of execution of the parallel goal is *trapped*, i.e., it is currently not at the top of the stack. In this case, there are two possible scenarios. If the goal is known not to have additional solutions, for instance because it did not push any choice point or because it has been marked as deterministic during compilation, or by the user [BLGPH06, HPBG05], then the segment where the goal lies does not need to be expanded and the pointers to the top of the segment in the handler are simply made to point to the beginning of the segment. The section of the trail corresponding to that segment is used to undo the bindings. After this the stack and the trail pointers are restored to their previous values — i.e., they point to the tops of the corresponding stacks.

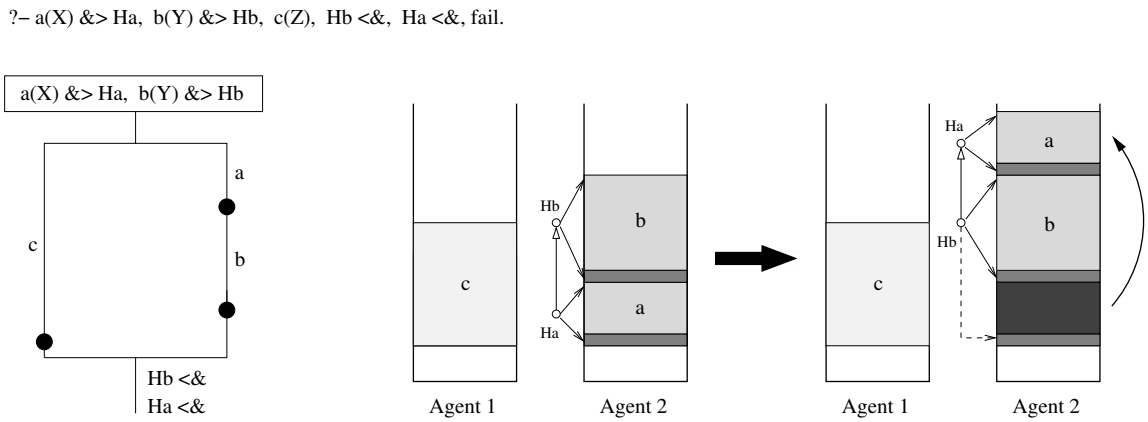


Figure 5.7: Copying trapped goal onto top of stack.

If there may be more solutions for that goal, then a mechanism is needed to untrap the segment of execution of the goal. Several solutions have been proposed to solve this problem [Her87, SH96]. A first approach consists of avoiding it altogether by carefully selecting goals to be executed so that they cannot cause trapped goals (which would dramatically reduce the amount of exploited parallelism). Another solution is to create a new, independent stack set for every goal taken, which would probably be memory-inefficient or impose an extra overhead in memory management.

This proposal is a variant of the solution adopted by several parallel systems (e.g., ACE, DASWAM, &-Prolog, ...), which essentially try to continue the goal execution on top of the stack. However, in this case, and for simplicity, when a trapped goal is to be backtracked over, its execution segment is *copied* on top of the stack, where it can expand freely. The garbage slot created is marked as such, and can be recovered when everything between this garbage slot and the top of the stack turns into garbage (or on backtracking). Figure 5.6(e) shows how the limits of the segment of execution of the parallel goal are stored in the handler, so their values can be accessed in backwards execution, through the `save_init_execution/1` and `save_end_execution/1` primitives, which actually have similar behavior to that of

the input markers and end markers in the &-Prolog model. Note that the choice point created by the predicate `call_handler/1` is in fact the input marker of the parallel execution, but again defined in the source language. Finally, when the goal execution fails, the `metacut_garbage_slots/1` primitive will pop from the stack those discarded segments that are right underneath the segment of execution of the parallel goal. All these primitives extend the list of primitives shown in Section 5.1.1.

Although most of the garbage collection implementations do not recover dead choice points, this does not affect the presentation of the implementation, since the garbage collection algorithms have to be changed anyway to work with parallel execution and cross-agent pointers. Handlers already keep pointers to boundaries of every live segment, which the improved garbage collector algorithm can use.

Figure 5.7 shows an example of this solution for the trapped goal and garbage slot problems. It is assumed that the variables `X`, `Y`, and `Z` are independent. When literals `a/1` and `b/1` are taken and executed by the second agent, the pointers that define the actual segment of execution of both literals are stored in the corresponding handler. Thus, when `Ha <&` is reached in backtracking, the segment of execution of literal `a/1` is trapped, and it is copied on top of the stack in order to have enough space to expand and obtain a new solution for the goal `a/1`. The handler associated to the literal `b/1` will in addition mark the garbage slot left by the literal `a/1`, which will be freed when the execution of the literal `b/1` fails.

Figure 5.8 presents a state diagram which shows the different states in which a parallel goal can be, and graphically represents the and-parallel execution of goals previously shown in Figure 5.6. First, a goal is published to be executed in parallel, by adding a pointer to it in the goal list and releasing the execution of an agent that is currently idle. When performing the goal join, if the goal is still available in the goal list it will be executed locally. If the goal was found by some other agent then it

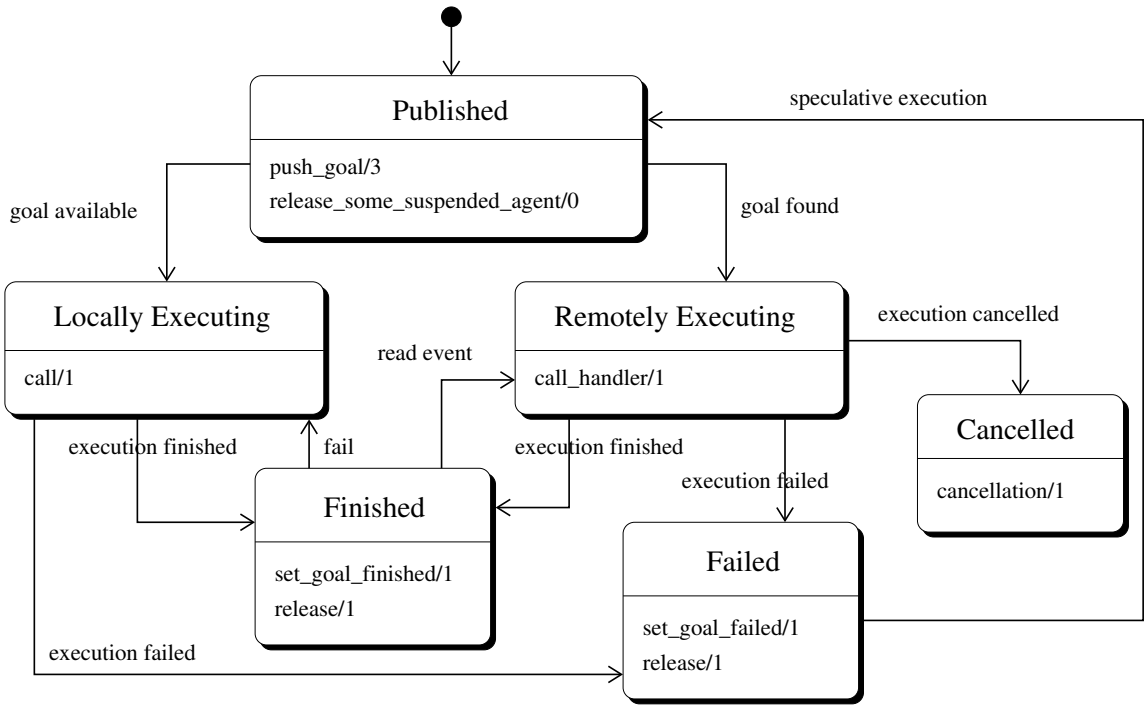


Figure 5.8: State diagram of a parallel goal.

will be executed remotely. That goal execution could be cancelled if the outcome of the execution is not needed for the actual computation. If the goal execution is not cancelled then it may succeed, in which case it may be backtracked over with the communication between agents performed via adding and removing events, or fail, in which case the goal will be published again for parallel execution.

5.3 Experimental Results

This section presents the performance results obtained after executing a selection of well-known benchmarks with independent and-parallelism. Although most of these benchmarks are quite well-known, Table 5.1 provides a brief description of them.

AIAKL	An abstract interpreter for the AKL language.
Ann	Annotator for and-parallelism.
Boyer	Simplified version of the *Boyer-Moore* theorem prover.
Chat	Question parser of *Chat-80*.
Deriv	Symbolic derivation.
FFT	Fast Fourier transform.
Fibonacci	Doubly recursive *Fibonacci*.
Hamming	Calculates *Hamming* numbers.
Hanoi	Solves *Hanoi* puzzle.
MergeSort	Sorts a $10,000$ element list.
MMatrix	Multiplies two 50×50 matrices.
Numbers	Obtains a number from a list of others.
Palindrome	Generates a palindrome of 2^{14} elements.
Progeom	Constructs a perfect difference set of order n.
Queens	The *n-queens* problem.
QueensT	Solves the *n-queens* problem T times.
QuickSort	Sorts a $10,000$ element list.
Takeuchi	Computes *Takeuchi*.

Table 5.1: Benchmarks to measure the performance of the high-level IAP implementation.

As mentioned before, the proposed approach has been implemented in the Ciao system [BCC+06], and all the benchmarks in Table 5.1 were automatically parallelized [MBdlBH99] using CiaoPP [HPBG05], and starting from their sequential code. All results were obtained by averaging ten runs on a state-of-the-art multiprocessor, a Sun Fire T2000 with 8 cores (4 threads each), and 8 Gb of memory, the same machine used to evaluate the annotation algorithms presented in Chapter 4.

While each core is capable in theory of running 4 threads in parallel, and in theory up to 32 threads could run simultaneously on this machine, only speedups up to 8 agents are shown. The Sun Fire T2000 machine cannot produce linear speedups beyond 8 processors even for independent computations due to the limitations in

the hardware of the multiprocessor machine[3]. Thus, beyond 8 agents, it is hard to know whether reduced speedups are due to the parallelization and implementation or to limitations of the machine. In order to experiment with this issue, a Prolog example using as many threads as natively available in the machine was executed, and its speedup compared to that of a C program generating completely independent computations. Such C program returned a practical upper bound on the attainable speedups. The results are depicted in Figure 5.9, which shows both the ideally parallel C program and a parallelized Fibonacci running on the implementation. Interestingly, the speedup obtained is only marginally worse than the best possible one. In both curves it is possible to observe a sawtooth shape, presumably caused by tasks filling in a row of units in all cores and starting to use up additional thread units in other cores, which happens at 1×8, 2×8, and 3×8 threads.

Table 5.2 presents the speedups obtained after running only deterministic programs (using `&!/2`, `&!>/2` and `<&!/1`) parallelized using [N]SIAP. Table 4.2 already presented a comparison of the performance results for some of the benchmarks annotated with restricted and unrestricted IAP. The speedups are with respect to the sequential speed on one processor of the original, not parallelized benchmark. Therefore, the column tagged *1* corresponds to the slowdown coming from executing a parallel program on a single processor. Benchmarks with a *GC* suffix were executed with granularity control with a suitably chosen threshold and benchmarks with a *DL* suffix use difference lists (hence no `append/3` is needed), and require NSIAP for parallelization. All the benchmarks were automatically parallelized using CiaoPP [HPBG05] and the annotation algorithms described in [CCH07a].

It can be deduced from the results that in several benchmarks the *natural* parallelizations produce small granularity. This, understandably, impacts the imple-

[3]Mainly, we suspect the availability of a reduced number of integer units and a single FP unit.

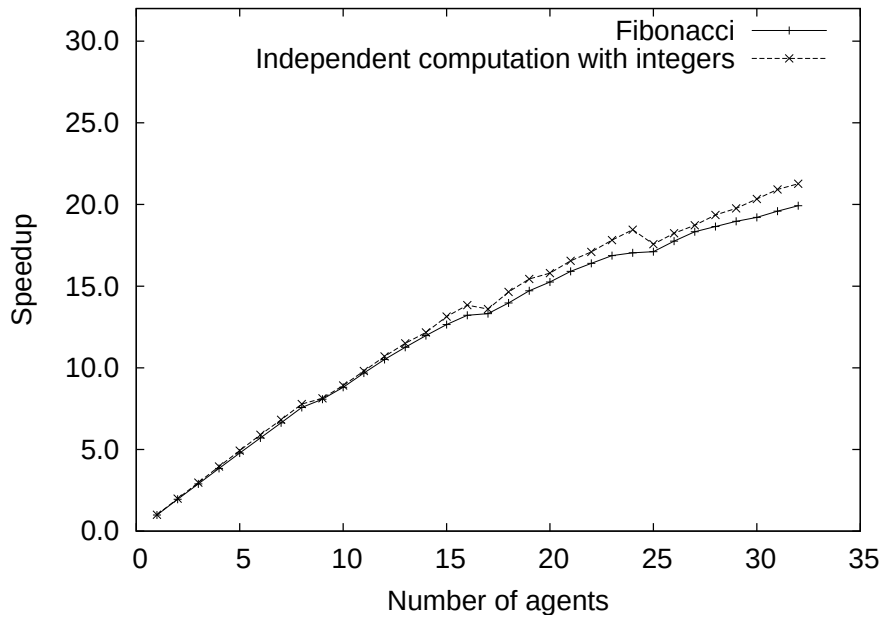


Figure 5.9: Performance results of *Fibonacci* with granularity control vs. maximum speedup in real machine (Sun Fire T2000, 8 cores, 8 Gb of memory and 4 threads per core).

mentation since a sizable part of it is written in Prolog, which implies additional overhead in the preparation and execution of parallel goals. Thus, it is not possible to perform a fair comparison of the speedups obtained with respect to previous (lower-level) and-parallel systems. In addition, &-Prolog does not run in current multiprocessors machines and, since these machines are not as "parallel" as, e.g., early shared-memory multiprocessors, such as the Sequent Balance or Symmetry machines, where early parallel logic programming systems were benchmarked, performance results cannot be compared directly.

The overhead implied by the proposed approach produces comparatively low performance on a single processor, and in some cases with very fine granularity, such as *Boyer* and *Takeuchi*, speedups are shallow (below 2×) even over 8 processors. In these examples, execution is dominated by the sequential code of the scheduler and

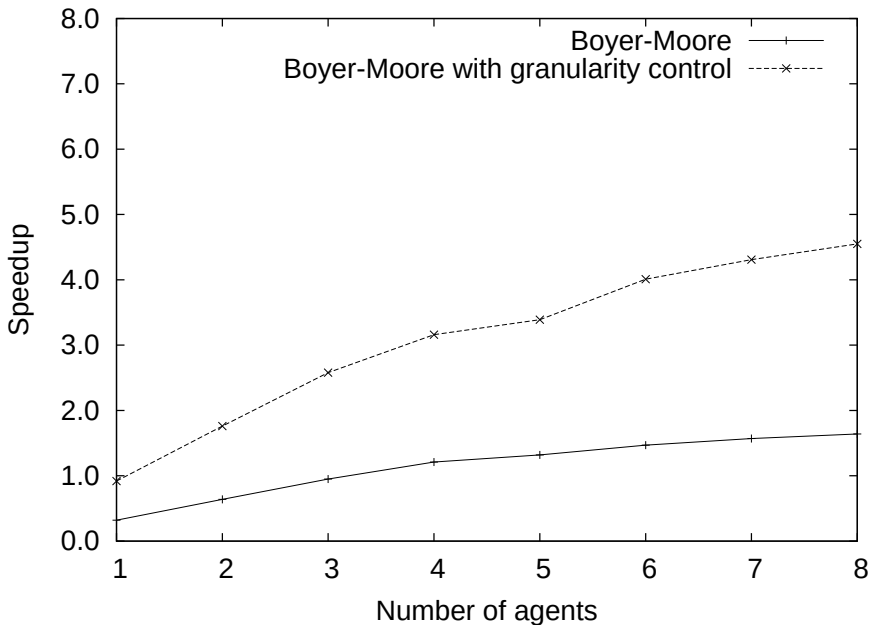


Figure 5.10: Speedups obtained with and without granularity control for *Boyer*.

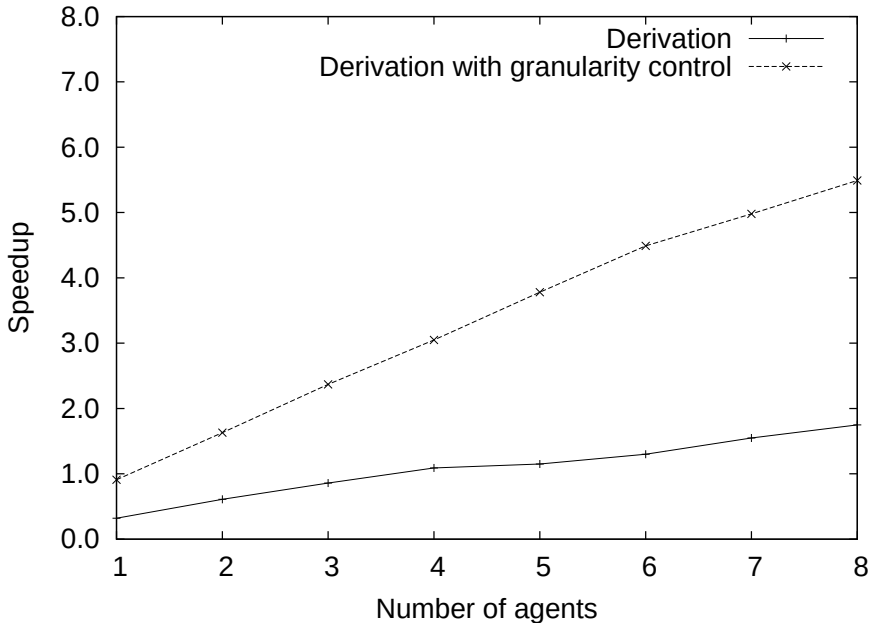


Figure 5.11: Speedups obtained with and without granularity control for *Deriv*.

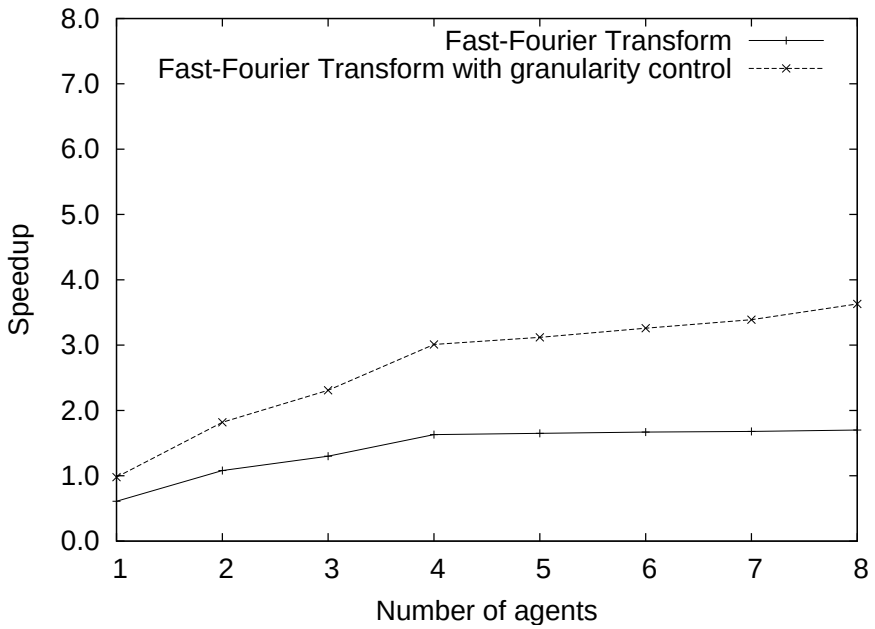


Figure 5.12: Speedups obtained with and without granularity control for *FFT*.

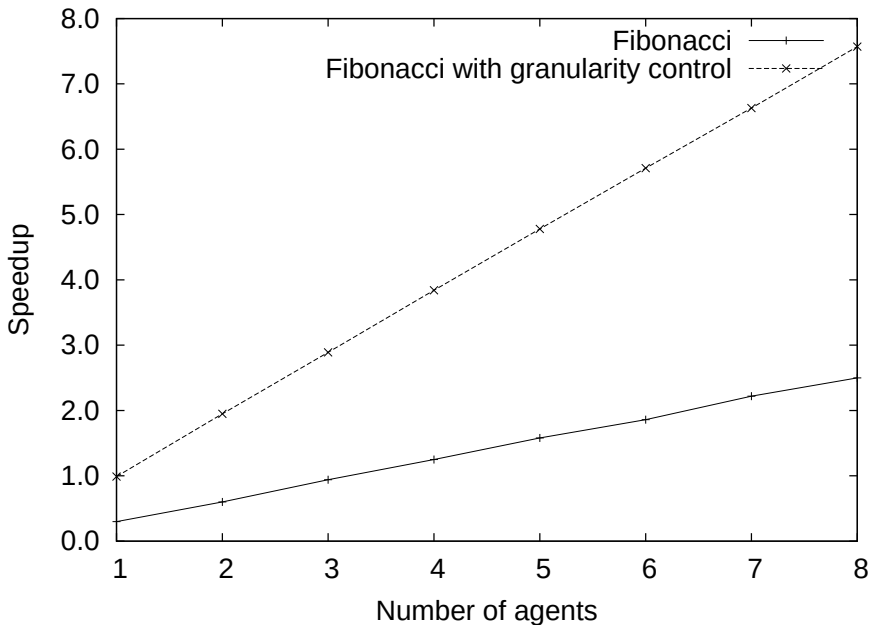


Figure 5.13: Speedups obtained with and without granularity control for *Fibonacci*.

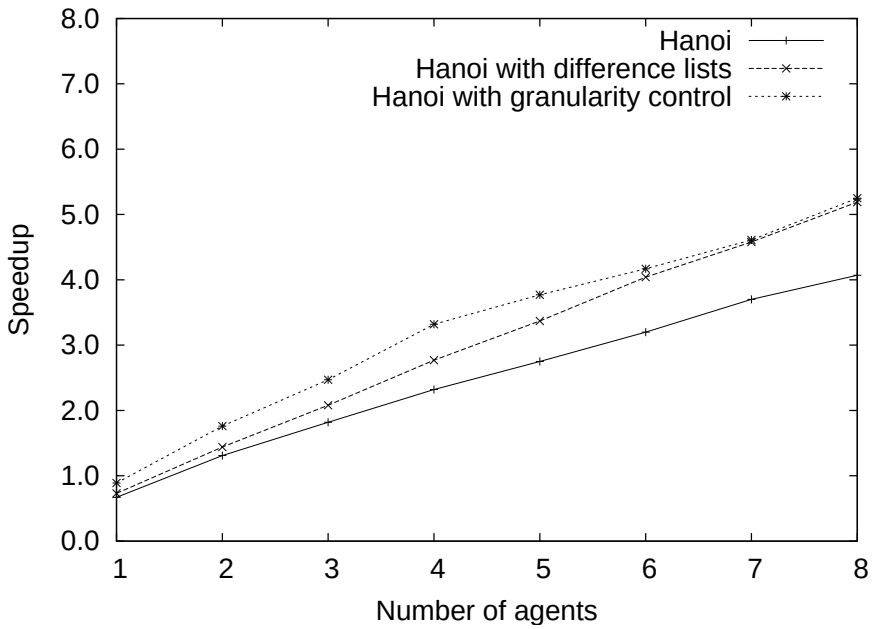


Figure 5.14: Speedups obtained with and without granularity control for *Hanoi*.

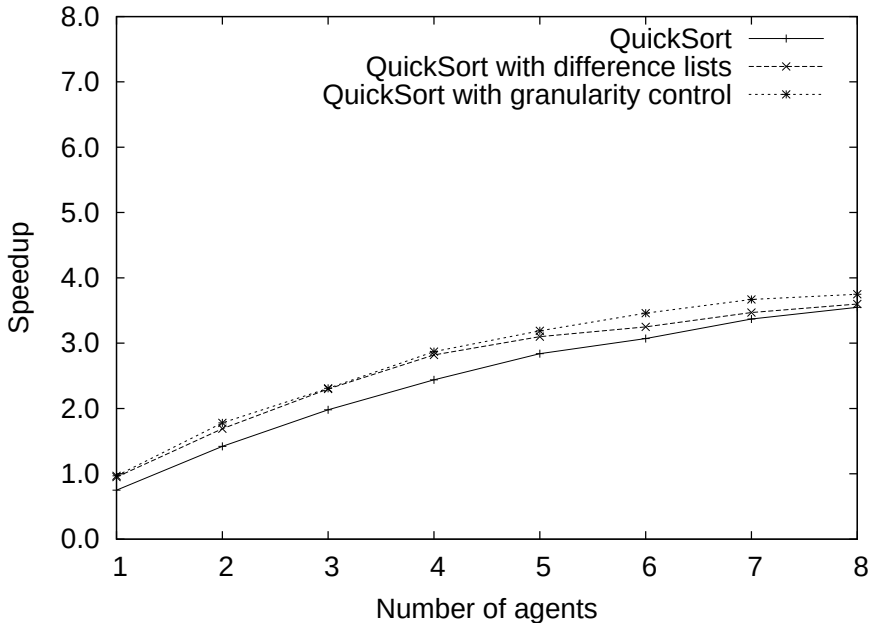


Figure 5.15: Speedups obtained with and without granularity control for *QuickSort*.

Benchmark	**Number of processors**							
	1	2	3	4	5	6	7	8
AIAKL	0.97	1.82	1.82	1.82	1.82	1.83	1.83	1.83
Ann	0.98	1.86	2.72	3.56	4.38	5.16	5.88	6.64
Boyer	0.32	0.64	0.95	1.21	1.32	1.47	1.57	1.64
BoyerGC	0.92	1.76	2.58	3.16	3.39	4.01	4.31	4.55
Deriv	0.32	0.61	0.86	1.09	1.15	1.30	1.55	1.75
DerivGC	0.91	1.63	2.37	3.05	3.78	4.49	4.98	5.49
FFT	0.61	1.08	1.30	1.63	1.65	1.67	1.68	1.70
FFTGC	0.98	1.82	2.31	3.01	3.12	3.26	3.39	3.63
Fibonacci	0.30	0.60	0.94	1.25	1.58	1.86	2.22	2.50
FibonacciGC	0.99	1.95	2.89	3.84	4.78	5.71	6.63	7.57
Hamming	0.93	1.15	1.64	1.64	1.64	1.64	1.64	1.64
Hanoi	0.67	1.31	1.82	2.32	2.75	3.20	3.70	4.07
HanoiDL	0.73	1.44	2.08	2.77	3.37	4.04	4.58	5.19
HanoiGC	0.89	1.76	2.47	3.32	3.77	4.17	4.61	5.25
MergeSort	0.79	1.47	2.12	2.71	3.01	3.30	3.56	3.71
MergeSortGC	0.83	1.52	2.23	2.79	3.10	3.43	3.67	3.95
MMatrix	0.91	1.74	2.55	3.32	4.18	4.83	5.55	6.28
Palindrome	0.44	0.77	1.09	1.40	1.61	1.82	2.10	2.23
PalindromeGC	0.96	1.79	2.37	2.97	3.30	3.62	4.13	4.46
QuickSort	0.75	1.42	1.98	2.44	2.84	3.07	3.37	3.55
QuickSortDL	0.95	1.69	2.30	2.82	3.10	3.25	3.47	3.60
QuickSortGC	0.97	1.78	2.31	2.87	3.19	3.46	3.67	3.75
Takeuchi	0.23	0.46	0.68	0.91	1.12	1.32	1.49	1.72
TakeuchiGC	0.88	1.62	2.39	3.33	4.04	4.47	5.19	5.72

Table 5.2: Speedups obtained for several deterministic IAP benchmarks.

agent management in Prolog. However, even in these cases, setting a granularity threshold based on a measure of the input argument size [LGHD96] much better results can be obtained. Figures 5.10, 5.11, 5.12, 5.13, 5.14 and 5.15 depict graphically the impact of granularity control in some of the selected benchmarks. Annotating the parallelized program to take into account granularity measures based on size of the input arguments, and automatically finding out the optimal threshold for a given platform, can be done automatically in many cases [LGHD96, MLGP+07].

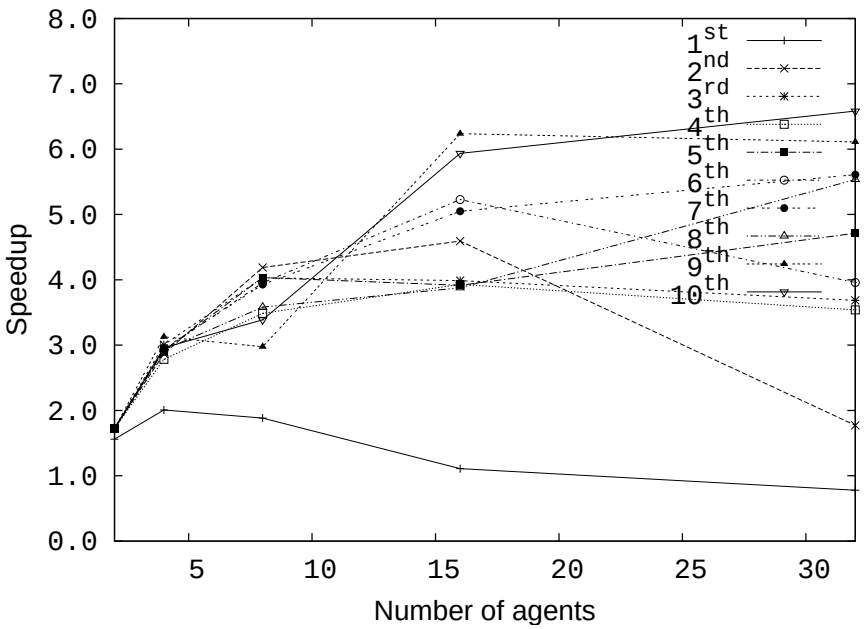


Figure 5.16: Speedups with stack set expansion for *Boyer*.

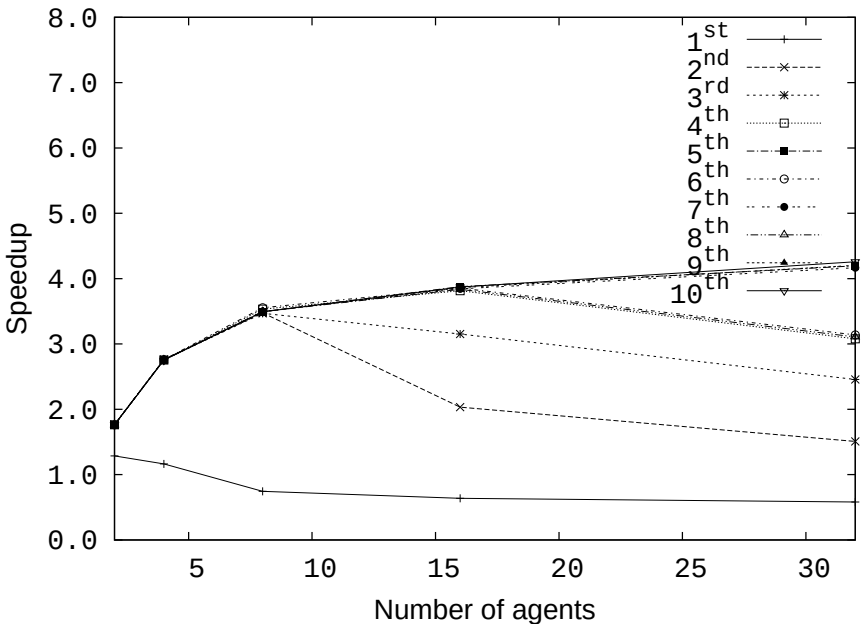


Figure 5.17: Speedups with stack set expansion for *FFT*.

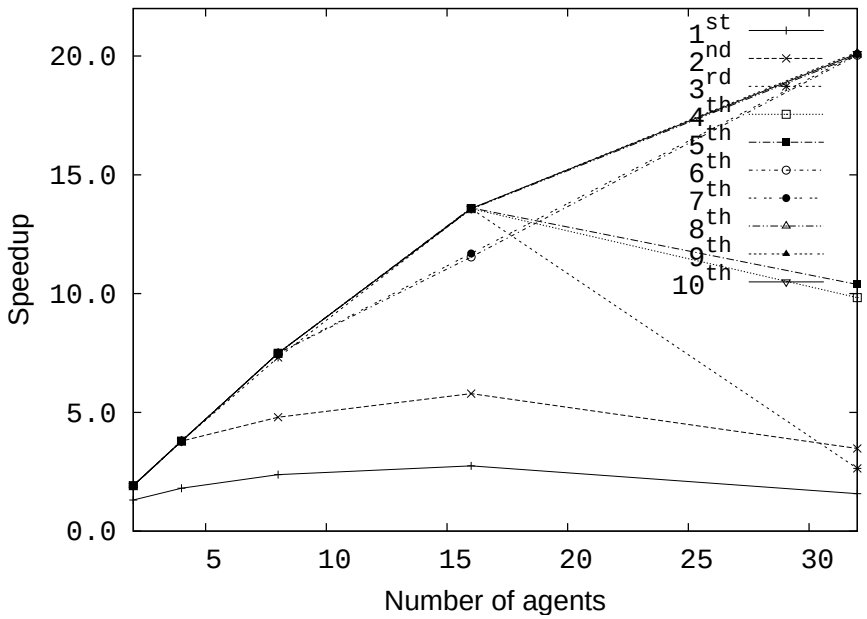


Figure 5.18: Speedups with stack set expansion for *Fibonacci*.

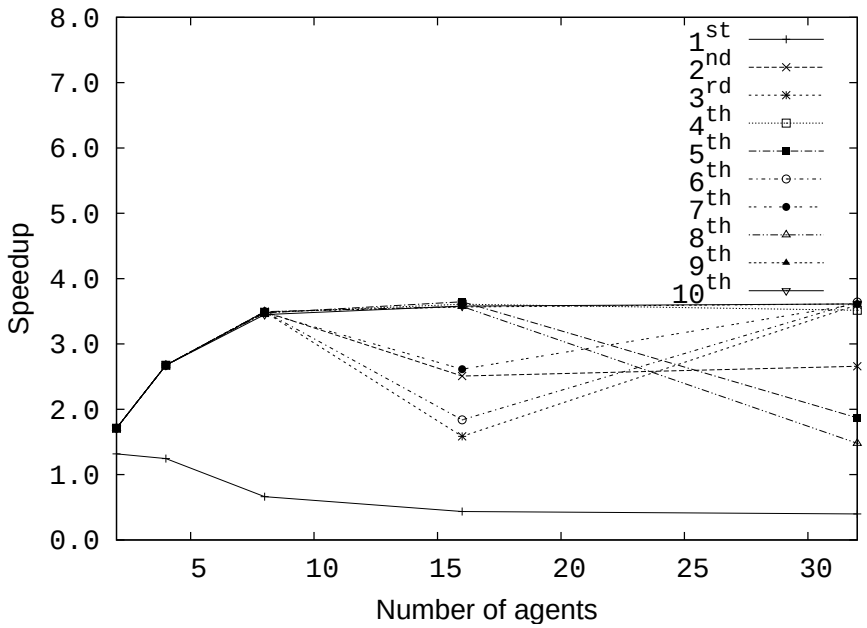


Figure 5.19: Speedups with stack set expansion for *QuickSort*.

Benchmark	**Op.**	**Number of agents**								
		Seq.	1	2	3	4	5	6	7	8
AIAKL	&!	1.00	0.97	1.82	1.82	1.82	1.83	1.83	1.83	1.82
	&	1.00	0.96	1.70	1.71	1.72	1.74	1.75	1.72	1.72
Ann	&!	1.00	0.98	1.86	2.72	3.56	4.38	5.16	5.88	6.64
	&	1.00	0.96	1.85	2.72	3.57	4.35	5.14	5.87	6.61
Boyer	&!	1.00	0.92	1.76	2.58	3.16	3.39	4.01	4.31	4.55
	&	1.00	0.90	1.21	1.83	2.06	2.26	2.30	2.39	2.56
Deriv	&!	1.00	0.91	1.63	2.37	3.05	3.78	4.49	4.98	5.49
	&	1.00	0.84	1.60	2.34	2.99	3.73	4.43	4.56	4.85
FFT	&!	1.00	0.98	1.82	2.31	3.01	3.12	3.26	3.39	3.63
	&	1.00	0.98	1.72	1.97	2.65	2.67	2.75	2.93	2.97
Fibonacci	&!	1.00	0.99	1.95	2.89	3.84	4.78	5.71	6.63	7.57
	&	1.00	0.98	1.58	2.04	2.53	3.28	4.06	4.61	5.46
Hamming	&!	1.00	0.93	1.15	1.64	1.64	1.64	1.64	1.64	1.64
	&	1.00	0.92	1.02	1.41	1.63	1.62	1.62	1.62	1.62
Hanoi	&!	1.00	0.89	1.76	2.47	3.32	3.77	4.17	4.61	5.25
	&	1.00	0.89	1.77	1.91	2.84	3.13	3.54	3.96	4.47
HanoiDL	&!	1.00	0.73	1.44	2.08	2.77	3.37	4.04	4.58	5.19
	&	1.00	0.74	1.43	1.89	1.87	2.73	3.07	3.59	3.87
MMatrix	&!	1.00	0.91	1.74	2.55	3.32	4.18	4.83	5.55	6.28
	&	1.00	0.90	1.48	2.16	2.88	3.51	4.13	4.71	5.25
Palindrome	&!	1.00	0.96	1.79	2.37	2.97	3.30	3.62	4.13	4.46
	&	1.00	0.96	1.78	2.14	2.56	3.11	3.30	3.74	3.90
QuickSort	&!	1.00	0.97	1.78	2.31	2.87	3.19	3.46	3.67	3.75
	&	1.00	0.97	1.71	2.17	2.43	2.60	2.93	3.06	3.19
QuickSortDL	&!	1.00	0.95	1.69	2.30	2.81	3.10	3.25	3.47	3.60
	&	1.00	0.95	1.68	2.14	2.39	2.56	2.92	2.94	3.19
Takeuchi	&!	1.00	0.88	1.62	2.39	3.33	4.04	4.47	5.19	5.72
	&	1.00	0.88	1.45	2.02	2.85	3.41	3.80	4.23	4.66

Table 5.3: Speedups obtained for several deterministic unrestricted IAP benchmarks.

Another particular fact that limits the performance results is the expansion of the stack set of an agent when space allocated to them is about to be exhausted. Stack sets are initially created with a relatively small size, and they dynamically grow as

Benchmark	**Number of agents**								
	Seq.	1	2	3	4	5	6	7	8
Chat	1.00	2.31	4.49	5.42	6.91	9.79	9.95	11.10	17.29
Numbers	1.00	1.84	1.79	1.79	1.79	1.79	1.79	1.78	1.78
Progeom	1.00	0.99	0.96	0.97	0.98	0.98	0.98	0.98	0.98
Queens	1.00	0.99	0.94	0.94	0.94	0.94	0.94	0.94	0.94
QueensT	1.00	0.99	1.90	2.41	3.18	4.71	4.61	4.58	4.57

Table 5.4: Speedups obtained for several non-deterministic unrestricted IAP benchmarks.

needed. Due to the work-stealing strategy adopted, and the shared-memory nature of this implementation, there are cross-agent pointers. The approach that has been taken in this prototype implementation to ensure a correct stack set expansion is to suspend the execution of all the agents. The stack set which is short on space is then expanded and the pointers pointing to that stack set (from any agent) are updated. The execution of the agents finally resumes. A smarter algorithm could be implemented, but this topic is out of the scope of this thesis and a subject for further work.

That scheme indeed affects the performance of the execution. Figures 5.16, 5.17, 5.18 and 5.19 present the speedups obtained when executing some selected benchmarks with 2, 4, 8, 16 and 32 agents, ten consecutive times. By joining together the points corresponding to the n-th execution with a given number of processors, we can construct a profile of how the speedup evolves as the system executes several times the same programs. It can be observed that the first executions do suffer from stack expansions. However, after several executions, the stack set of each agent reaches an appropriate size, reducing the number of expansions, and thus the performance results stabilize.

Table 5.3 presents the speedups obtained for some deterministic benchmarks par-

		Benchmark											
		Queens, 2 agents				Queens, 4 agents				Queens, 8 agents			
		No		Gr		No		Gr		No		Gr	
		1	N	1	N	1	N	1	N	1	N	1	N
`G &> H`		11,810	171,858	9	290	11,810	171,858	9	290	11,810	171,858	9	290
Taken	$\overline{x}$	6,649	97,798	9	290	6,860	99,373	9	290	6,476	96,056	9	290
	σ	9.35	45.04	0.00	0.00	16.15	65.02	0.00	0.00	13.49	59.04	0.00	0.00
LBack	$\overline{x}$	858	14,319	0.00	0.00	618	10,905	0.00	0.00	755	12,786	0.00	0.00
	σ	1.03	1.25	0.00	0.00	14.93	99.89	0.00	0.00	5.79	23.59	0.00	0.00
RBack Top	$\overline{x}$	1,838	29,725	2	234	2,345	38,420	2	234	2,208	36,261	2	234
	σ	0.46	2.14	0.00	0.00	15.14	98.66	0.00	0.00	6.34	26.53	0.00	0.00
RBack Tp	$\overline{x}$	0	0	0	0	0	0	0	0	0	0	0	0
	σ	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 5.5: Behavior of `Queens(8)` with different number of agents.

		Benchmark											
		Progeom, 2 agents				Progeom, 4 agents				Progeom, 8 agents			
		No		Gr		No		Gr		No		Gr	
		1	N	1	N	1	N	1	N	1	N	1	N
`G &> H`		215	154,260	1	60	215	154,260	1	60	215	154,260	1	60
Taken	$\overline{x}$	100	72,375	0	1	91	65,643	0	1	55	75,113	0	1
	σ	1.85	248.69	0.00	0.80	1.36	414.68	0.00	0.70	3.49	192.25	0.00	0.78
LBack	$\overline{x}$	1	738	0	29	3	2,131	0	29	9	364	0	29
	σ	0.46	52.03	0.00	0.80	1.10	83.78	0.00	0.70	0.80	26.82	0.00	0.78
RBack Top	$\overline{x}$	10	6,530	0	1	8	5,131	0	1	2	6,907	0	1
	σ	0.57	52.08	0.00	0.80	1.10	84.26	0.00	0.70	0.80	27.02	0.00	0.78
RBack Tp	$\overline{x}$	0	0	0	0	0	0	0	0	0	0	0	0
	σ	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 5.6: Behavior of `Progeom(5)` with different number of agents.

allelized using unrestricted IAP, and executed with both deterministic (`&!/2`) and non-deterministic operators (`&/2`). As well as in Table 5.2, the speedups were obtained with respect to the execution time that the sequential version of the benchmarks takes on one processor.

It can be concluded from the results that the difference in speedups between both parallel versions is of little significance in most cases, and only in very few cases (for example, `Boyer` and `Fibonacci`) the difference is relevant. Note that determinism can either be annotated by hand or, in many cases, automatically detected by a

		Benchmark											
		Fibonacci, 2 agents				Fibonacci, 4 agents				Fibonacci, 8 agents			
		No		Gr		No		Gr		No		Gr	
		1	N	1	N	1	N	1	N	1	N	1	N
`G &> H`		121,392	121,392	1,596	1,596	121,392	121,392	1,596	1,596	121,392	121,392	1,596	1,596
Taken	$\overline{x}$	1	1	1	1	5	5	5	5	37	37	31	31
	σ	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	3.97	3.97	2.39	2.39
LBack	$\overline{x}$	121,391	121,391	1,595	1,595	121,387	121,387	1,591	1,591	121,355	121,355	1,565	1,565
	σ	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	3.97	3.97	2.39	2.39
RBack Top	$\overline{x}$	1	1	1	1	5	5	5	5	18	18	16	16
	σ	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	2.40	2.40	0.98	0.98
RBack Tp	$\overline{x}$	0	0	0	0	0	0	0	0	19	19	15	15
	σ	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	2.86	2.86	1.68	1.68

Table 5.7: Behavior of `Fibonacci(25)` with different number of agents.

sophisticated analyzer [BLGPH06, HPBG05]. In any case, reasonably good speedups are obtained, despite the fact that the proposal suffers from the overhead added by the source-level coded scheduler etc., but which, in return, offers other advantages such as significantly reduced development (and maintenance) time, more flexibility, simpler and faster experimentation, etc.

Table 5.4 presents the speedups obtained for some non-deterministic benchmarks executed in parallel. Some of them do not achieve any speedup when executed in parallel due to the very fine granularity of the parallel goals in these benchmarks and the high-level nature of this implementation. However, super-linear speedups can be achieved in some other benchmarks, thanks to the implementation of goal cancellation.

Tables 5.5, 5.6 and 5.7 present some data from the execution of some of the benchmarks which exploit and-parallelism on non-deterministic programs.

They present the data from executions with 2, 4, and 8 agents, using or not granularity control (resp., `Gr` and `No`), and for the cases in which only one solution (`1`) or all solutions (`N`) are requested. The first row in the table (`G &> H`) contains the number of parallel goals published. The second row (*Taken*) presents the number

of parallel goals that were picked up by some other agent. $\overline{x}$ is always the average of the results in ten runs and σ will correspond to the standard deviation. The third row (*LBack*) represents the number of times that backtracking over parallel goals took place locally (because the goal was not picked up by some other agent).[4] The fourth row in the table (*RBack*) shows the number of times that a parallel goal was backtracked over remotely. In this case, *Top* counts how many times remote backtracking was performed at the top of the stack, and `Tp` shows the number of times that backtracking over a trapped goal was necessary. A relevant conclusion extracted from these results is that, while the amount of remote backtracking is quite high, the number of trapped goals is surprisingly low. Therefore, the overhead associated with copying the trapped segments to the top of the stack should not be very expensive in comparison with the rest of the goal execution.

Furthermore, it is expected to see a similar behavior in most of the non-deterministic parallel programs where parallel goals are of very fine granularity or very likely to fail: these two behaviors make the piling up of segments corresponding to the execution of loosely related parallel goals in the same stack relatively uncommon, which indeed reduces the chances to suffer from the trapped goal and garbage slot problems.

5.4 Summary

A new implementation approach for exploiting independent and-parallelism in logic programs has been presented, with the objectives of providing a simpler machinery and more flexibility than previous approaches. The approach is based on raising the

[4]The backtracking measured for `Fibonacci` in Table 5.7 corresponds to the stack unwinding performed when failing after the execution is finished.

implementation of some components to the source language level by using more basic high-level primitives than the fork-join operator and keeping only some relatively simple operations at a lower level, making the system easier to code, maintain, and extend. The high-level execution model for unrestricted IAP presented in this chapter has been implemented and evaluated in the Ciao system.

Part IV

Conclusions

Chapter 6

Concluding Remarks and Future Work

This final chapter is intended to conclude the thesis by presenting a brief overview of the main contributions made, situated in the domain of automatic parallelization of multiparadigm declarative languages. Furthermore, some suggestions for future research will be made.

6.1 Functional Notation and Lazy Evaluation in LP Systems

As previously mentioned in this thesis, the idea of adding functional features to logic programming systems is not new [BBLM84, BL86, Nai91], since there are currently a good number of systems which integrate functions and higher-order programming into some form of logic programming. However, the approach presented in Chapter 3 is very interesting in itself. First, the approach is completely syntactic, and thus

functions are mere predicates, thus maintaining their power. Also, predicates may be invoked through functional syntax or evaluated lazily. In addition, this syntactic extension of the language is composable with other extensions such as higher-order or objects. Moreover, another important characteristic of this approach is that most of it can be applied directly (or with minor changes) to any ISO-standard Prolog system.

Finally, and perhaps most importantly, and again because of the syntactic nature of this extension, it can be the target of analysis, optimization, static checking, and verification (of types, modes, determinacy, nonfailure, cost, etc.), as performed by, e.g., the Ciao preprocessor (CiaoPP) [HPBLG05].

The original version of the functional extension was first distributed in Ciao 0.2 [BCC+97] and later used as an example in [CH00]. As part of the evaluation of this extension, the performance of the lazy evaluation was tested with several examples. As expected, the usage of lazy evaluation implies some extra time and memory overhead, which justifies making lazy evaluation optional to the user via a declaration.

6.2 Automatic Unrestricted Annotation for IAP

Chapter 4 presented and proved correct two different annotation algorithms for IAP which rely on the use of high-level primitives that have a simpler nature than the traditional fork-join operator. The main difference between the algorithms is the imposed restriction to preserve the order of the solution in the original or not.

The annotation algorithms have been implemented and evaluated in CiaoPP, using the high-level execution model for IAP presented in Chapter 5. The results

of the experiments performed show that, although the parallelization provided by the new annotation algorithms is the same in quite a few of the traditional parallel benchmarks, in the experiments it is actually never worse and in some cases it is significantly better. Some simple representative cases were presented in which the performance results were notoriously increased by the unrestricted parallelization. This supports the observations made based on the expected performance of the annotations.

In addition, the benefits are larger for programs with high numbers of goals in their clauses, since their more complex graphs make the ability to exploit unrestricted parallelism more relevant.

6.3 High-Level Execution Model for Unrestricted IAP

Most of the previous implementations of and-parallelism have relied on a very complex low-level machinery, which was indeed very difficult to code and maintain. Therefore, to the best of my knowledge, no currently available low-level implementation of Prolog with and-parallelism fully implements backtracking of non-deterministic parallel goals.

Chapter 5 presented a new execution model, based on a high-level implementation, which is able to exploit full independent and-parallelism, both restricted and unrestricted. It has been implemented as part of the Ciao system, and evaluated with representative benchmarks, some of which perform backtracking over non-deterministic parallel goals. Furthermore, performance data from actual parallel executions has been shown in order to provide a better understanding of the

behavior of the parallel execution.

The experimental results show that quite reasonable speedups are achievable with this approach, although the additional overhead makes it necessary to use granularity control in many cases in order to obtain good performance results. While this might seem to complicate the code, in fact automatic, compile-time granularity analysis can be applied to alleviate the programmer from the burden of adding such control by hand.

In addition, for the non-deterministic parallel benchmarks, in several cases super-linear speedups were obtained thanks to the backtracking model implemented, since the internal failure of a goal in a parallel, non-deterministic conjunction cancels the other goal.

The fundamental claim of this work is that a high-level implementation can be usefully competitive with a lower level one in terms of "speed / complication". It is clear, at least with the current state-of-the-art compilers, that once the components of the system stabilize additional parts of the code (those which are a bottleneck) that are written in Prolog could be moved down to C, in order to increase the overall performance of the system. However, that is not currently planned, since the benefits may not surpass the added complexity and reduced flexibility. Also, it should be taken into account that, by maintaining the flexibility of the system, smarter schedulers could be, in principle, easier to develop than with other approaches.

But, most importantly, some of the recent compilation technology and implementation advances, as in [CMM+06, SC99, MCH07], provide hope that it will eventually be possible to recover a significant part of the efficiency lost due to the level at which the parallel execution is expressed simply because the difference in speed between the high-level (Prolog) code and C is diminishing significantly.

6.4 Future Work

There exist several improvements to the work presented in this thesis which could be developed and many directions in which it could be extended.

First, it would be very interesting to enrich the functional syntax extension to the language with more extensive higher-order programming features, as for instance the possibility to have higher-order lambda-term (pattern) unification. Due to the Ciao system's design features, the higher-order pattern unification could be used as an optional extension to the user, in order to avoid the performance overhead that other implementations, as for instance Lambda Prolog [NM88], have to pay for having it by default. In this direction, preliminary implementation was done and is currently a library in the Ciao system.

Also, since the performance results of the high-level solution for unrestricted IAP which have been obtained are very encouraging, further research could be done in a similar direction in order to improve the current state of the system and develop new ways of parallelizing logic programs. Both the UUDG and UOUDG annotation algorithms only take into account dependency information in order to decide whether a literal can be scheduled for parallel execution or not. These annotation algorithms could benefit from the use of cost and resource compile-time analysis [NMLGH07] in order to control the additional inherent overhead due to the nature of the high-level implementation. Some work has already been performed on designing new heuristics in order to perform approximate comparisons of cost functions at compile-time, as well as on extending the annotation algorithms presented in Chapter 4 to generate conditional expressions in order to provide different parallelizations depending on run-time evaluation of selected cost functions.

In addition, the high-level implementation of unrestricted IAP could be improved

by the usage of other existing tools, as for instance tabling [dGCH+08]. Also, developing new high-level operators would most likely allow the system to exploit new sources of parallelism. Finally, another issue for further research may be the design and evaluation of new efficient stack expansion and parallel garbage collection algorithms, in which the synchronization required between threads is minimized.

Part V

Appendices

Appendix A

Comparison Between Restricted and Unrestricted IAP

In this appendix, let us consider the predicate `p/3` of Section 4.1.1, whose dependency graph is shown in Figure 4.1. The following sections will show how the unrestricted parallelization is a better option in this case than the restricted ones.

A.1 Overhead in Parallel Execution Assumed to be Zero

The execution time expressions for the two restricted parallelizations of the predicate `p/3`, given in Figure 4.2(a) and Figure 4.2(b), are presented in Equation (4.1) and Equation (4.2), assuming that there is not overhead in the parallel execution. These two equations can be implemented using constraint logic programming (CLP) as shown in Figure A.1.

```
positive([]).
positive([X|Xs]) :-
        X .>. 0,
        positive(Xs).

%% Tfj1 = max(a + b, c) + d
tfj1(A, B, C, D, T) :-
        positive([A,B,C,D,T]),
        AB .=. A + B,
        max(AB, C, MaxABC),
        T .=. D + MaxABC.
```

```
max(X, Y, X) :- X .>=. Y.
max(X, Y, Y) :- X .<. Y.


%% Tfj2 = max(a, c) + max(b, d)
tfj2(A, B, C, D, T) :-
        positive([A,B,C,D,T]),
        max(A, C, MAC),
        max(B, D, MBD),
        T .=. MAC + MBD.
```

Figure A.1: CLP code for Equations (4.1) and (4.2), assuming no overhead in the parallel execution.

In order to compare Equations (4.1) and (4.2), each of the following queries are thrown:

```
?- tfj1(A,B,C,D,T1),
   tfj2(A,B,C,D,T2),
   T1 .<. T2.

yes
```

```
?- tfj1(A,B,C,D,T1),
   tfj2(A,B,C,D,T2),
   T2 .<. T1.

yes
```

Thus, none of the restricted parallelizations of predicate `p/3` is better than the other.

The parallelization resulting from the execution of the UUDG annotator is shown in Figure 4.3 and the expression which gives the execution time appears in Equation (4.3). In a similar fashion as in Figure A.1, Figure A.2 presents the constraint logic programming code that implements the unrestricted parallelization of predicate `p/3` shown in Equation (4.3).

```
%% Tnfj = max(a+b, d + max(a,c))
tnfj(A, B, C, D, T) :-
        positive([A,B,C,D,T]),
        AB .=. A + B,
        max(A, C, MaxAC),
        DAC .=. D + MaxAC,
        max(AB, DAC, T).
```

Figure A.2: CLP code for Equation (4.3), assuming no overhead in the parallel execution.

In order to compare Equation (4.3) with Equations (4.1) and (4.2), and decide whether any of the *fork-join* annotations can be handled better than the *non-fork-join* one with the `&>/2` and `<&/1` operators, the following queries can be executed:

```
?- tfj1(A,B,C,D,T1), tnfj(A,B,C,D,T2), T1 .<. T2.

no
?- tfj2(A,B,C,D,T1), tnfj(A,B,C,D,T2), T1 .<. T2.

no
```

Thus, it can be concluded that there does *not* exist a combination of execution times for the sequential goals that can make the *non-jork-join* annotation be worse than either of the *fork-join* ones. Therefore, Equation (4.3) will never perform worse than Equation (4.1) or Equation (4.2), and the unrestricted annotation is, therefore, a better option than any of the other restricted ones.

A.2 Considering Overhead in Parallel Execution

The previous section compared the restricted and unrestricted parallelization assuming that there was not overhead in the parallel execution. Figure A.3 presents the source code for both types of parallelization without that assumption. Focusing in the comparison between restricted and unrestricted parallelizations, the following queries can be executed:

```
?- tfj1_o(A,B,C,D,Oa,Ob,Oc,Oab,T1),
   tnfj_o(A,B,C,D,Oa,Ob,Oc,Oab,T2), T1 .<. T2.

Oa.>=.0,                    C.>.0,
C.=<.A,                     D.>.0,
D.=<.B,                     Oc.>.0,
D.<. -Oab+Ob+Oc,            Ob.>.0,
Oab.>.0,                    T2.=.B+Ob+Oc+A,
                            T1.=.D+Oab+B+A ?

yes
?- tfj2_o(A,B,C,D,Oa,Ob,Oc,Oab,T1),
   tnfj_o(A,B,C,D,Oa,Ob,Oc,Oab,T2), T1 .<. T2.

Oab.>=.0,                   D.>.0,
D.=<.B,                     Oa.>.0,
C.=<.A,                     C.>.0,
Oa.<.Oc,                    T2.=.Ob+A+Oc+B,
Ob.>.0,                     T1.=.Ob+Oa+A+B ?

yes
```

Thus, there are some cases in which the restricted parallelization may be better than the unrestricted one. By having a more in depth look at the answers, the restricted parallelization will take less time to execute if a particular goal takes less time to execute than a constant value, defined by the overhead of the parallelizations.

Therefore, it can be concluded that a notion of granularity control can be applied if the overhead of the parallelization is taken into account. Moreover, the simplification of the answers returned above will define the threshold that can be used to choose between parallelizations, in a similar fashion to the following:

```
p :-
        ...
        (
           granularity_threshold ->
           Unrestricted parallelization
        ;
           Restricted parallelization
        ),
        ...
```

```
%% Tfj1_o = max(a + b + oa, c) + d
tfj1_o(A, B, C, D, Oa, Ob, Oc, Oab, T) :-
        Oa .>=. 0,
        Ob .>=. 0,
        Oc .>=. 0,
        positive([A,B,C,D,Oab,T]),
        ABO .=. A + B + Oab,
        max(ABO, C, MaxABOC),
        T .=. D + MaxABOC.


%% Tfj2_o = max(a + oa, c) + max(b + ob, d)
tfj2_o(A, B, C, D, Oa, Ob, Oc, Oab, T) :-
        Oc  .>=. 0,
        Oab .>=. 0,
        positive([A,B,C,D,Oa,Ob,T]),
        AO .=. A + Oa,
        max(AO, C, MAOC),
        BO .=. B + Ob,
        max(BO, D, MBOD),
        T .=. MAOC + MBOD.


%% Tnfj_o = max(a + b, d + max(a,c)) + Ob + Oc
tnfj_o(A, B, C, D, Oa, Ob, Oc, Oab, T):-
        Oa  .>=. 0,
        Oab .>=. 0,
        positive([A,B,C,D,Ob,Oc,T]),
        AB .=. A + B,
        max(A, C, MaxAC),
        DAC .=. D + MaxAC,
        max(AB, DAC, MaxABDAC),
        T .=. MaxABDAC + Ob + Oc.
```

Figure A.3: CLP code for Equations (4.1), (4.2) and (4.3) taking into account the overhead of the parallel execution.

Bibliography

[AK90a] K. A. M. Ali and R. Karlsson. Full Prolog and Scheduling Or-parallelism in Muse. *International Journal of Parallel Programming*, 19(6):445–475, 1990.

[AK90b] K. A. M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.

[AK91] Hassan Ait-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction*. MIT Press, 1991.

[Ant91] S. Antoy. Lazy evaluation in logic. In *Symp. on Progr. Language Impl. and Logic Progr (PLILP'91)*, pages 371–382. Springer Verlag, 1991. LNCS 528.

[BBLM84] R. Barbuti, M. Bellia, G. Levi, and M. Martelli. On the integration of logic programming and functional programming. In *International Symposium on Logic Programming*, pages 160–168, Atlantic City, NJ, February 1984. IEEE Computer Society.

[BCC+97] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997. System and on-line version of the manual available at `http://www.ciaohome.org`.

[BCC+06] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School (UPM), 2006. Available at `http://www.ciaohome.org`.

[BdlBH94] F. Bueno, M. García de la Banda, and M. Hermenegildo. A Comparative Study of Methods for Automatic Compile-time Parallelization of Logic Programs. In *First International Symposium on Parallel Symbolic Computation, PASCO'94*, pages 63–73. World Scientific Publishing Company, September 1994.

[BdlBH99] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM Transactions on Programming Languages and Systems*, 21(2):189–238, March 1999.

[BdlBM+06] R. Becket, M. García de la Banda, K. Marriott, Z. Somogyi, P. J. Stuckey, and M. Wallace. Adding constraint solving to mercury. In *Eight International Symposium on Practical Aspects of Declarative Languages*, number 2819 in LNCS, pages 118–133. Springer-Verlag, January 2006.

[BHMR94] F. Bueno, M. Hermenegildo, U. Montanari, and F. Rossi. From Eventual to Atomic and Locally Atomic CC Programs: A Concurrent Semantics. In *Fourth International Conference on Algebraic and Logic Programming*, number 850 in LNCS, pages 114–132. Springer-Verlag, September 1994.

[BHMR98] F. Bueno, M. Hermenegildo, U. Montanari, and F. Rossi. Partial Order and Contextual Net Semantics for Atomic and Locally Atomic CC Programs. *Science of Computer Programming*, 30:51–82, January 1998. Special CCP95 Workshop issue.

[BL86] M. Bellia and G. Levi. The relation between logic and functional languages. *Journal of Logic Programming*, 3:217–236, 1986.

[BLGPH06] F. Bueno, P. López-García, G. Puebla, and M. Hermenegildo. A Tutorial on Program Development and Optimization using the Ciao Preprocessor. Technical Report CLIP2/06, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, January 2006.

[BLOO86] R. Butler, E. L. Lusk, R. Olson, and R. A. Overbeek. Anlwam: A Parallel Implementation of the Warren Abstract Machine. Internal report, Argonne National Laboratory, Argonne, Il 60439, 1986.

[BR86] P. Borgwardt and D. Rea. Distributed Semi-Intelligent Backtracking for a Stack-Based AND-Parallel Prolog. In *International Symposium on Logic Programming*, pages 211–222. IEEE Computer Society, 1986.

[BS81] 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.

[BSY88] P. Biswas, S. Su, and D. Yun. A Scalable Abstract Machine Model to Support Limited-OR/Restricted AND Parallelism in Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 1160–1179, Seattle,Washington, 1988.

[Cab04] D. Cabeza. *An Extensible, Global Analysis Friendly Logic Programming System*. PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, August 2004.

[Car87] M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the Wam. In *Fourth International Conference on Logic Programming*, pages 40–58. University of Melbourne, MIT Press, May 1987.

[CCH05] A. Casas, D. Cabeza, and M. Hermenegildo. Functional Notation and Lazy Evaluation in Ciao. In *Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS'05, ICLP associated workshop)*, pages 25–36, Sitges (Barcelona), October 2005.

[CCH06] A. Casas, D. Cabeza, and M. Hermenegildo. A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems. In *The 8th International Symposium on Functional and Logic Programming (FLOPS'06)*, pages 142–162, Fuji Susono (Japan), April 2006.

[CCH07a] A. Casas, M. Carro, and M. Hermenegildo. Annotation Algorithms for Unrestricted Independent And-Parallelism in Logic Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*, number 4915 in LNCS, pages 138–153, The Technical University of Denmark, August 2007. Springer-Verlag.

[CCH07b] A. Casas, M. Carro, and M. Hermenegildo. Towards A High-Level Implementation of Flexible Parallelism Primitives for Symbolic Languages. In *Parallel Symbolic Computation (PASCO'07)*, University of Western Ontario, July 2007. ACM Press. Extended Abstract.

[CCH07c] A. Casas, M. Carro, and M. Hermenegildo. Towards High-Level Execution Primitives for And-Parallelism: Preliminary Results. In *Colloquium on Implementation of Constraint and LOgic Programming Sys-*

tems (CICLOPS'07, ICLP associated workshop). U. of Evora, September 2007.

[CCH08a] A. Casas, M. Carro, and M. Hermenegildo. A High-Level Implementation of Non-Deterministic, Unrestricted, Independent And-Parallelism. In M. García de la Banda and E. Pontelli, editors, *24th International Conference on Logic Programming (ICLP'08)*, LNCS. Springer-Verlag, December 2008.

[CCH08b] A. Casas, M. Carro, and M. Hermenegildo. Towards a High-Level Implementation of Execution Primitives for Non-restricted, Independent And-parallelism. In D.S. Warren and P. Hudak, editors, *10th International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, volume 4902 of *LNCS*, pages 230–247. Springer-Verlag, January 2008.

[CDD85] J.-H. Chang, A. M. Despain, and D. Degroot. And-Parallelism of Logic Programs Based on Static Data Dependency Analysis. In *Compcon Spring '85*, pages 218–225. IEEE Computer Society, February 1985.

[CH83] A. Ciepielewski and S. Haridi. A formal model for or-parallel execution of logic programs. In P. Mason, editor, *Proceedings of IFIP*. North Holland, 1983.

[CH94] D. Cabeza and M. Hermenegildo. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. In *1994 International Static Analysis Symposium*, number 864 in LNCS, pages 297–313, Namur, Belgium, September 1994. Springer-Verlag.

[CH95] D. Cabeza and M. Hermenegildo. Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the 1995 COMPULOG-NET Workshop on Parallelism and Implementation Technologies*, Utrecht, NL, September 1995. U. Utrecht / T.U. Madrid. Available from http://www.cliplab.org/.

[CH96] D. Cabeza and M. Hermenegildo. Implementing Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the AGP'96 Joint conference on Declarative Programming*, pages 67–78, San Sebastian, Spain, July 1996. U. of the Basque Country. Available from `http://www.cliplab.org/`.

[CH99a] D. Cabeza and M. Hermenegildo. Higher-order Logic Programming in Ciao. Technical Report CLIP7/99.0, Facultad de Informática, UPM, September 1999.

[CH99b] D. Cabeza and M. Hermenegildo. The Ciao Modular Compiler and Its Generic Program Processing Library. In *ICLP'99 WS on Parallelism and Implementation of (C)LP Systems*, pages 147–164. N.M. State U., December 1999.

[CH00] D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.

[CH02] M. Carro and M. Hermenegildo. A simple approach to distributed objects in prolog. In *Colloquium on Implementation of Constraint and LOgic Programming Systems (ICLP associated workshop)*, Copenhagen, July 2002.

[CHL04] D. Cabeza, M. Hermenegildo, and J. Lipton. Hiord: A Type-Free Higher-Order Logic Programming Language with Predicate Abstraction. In *Ninth Asian Computing Science Conference (ASIAN'04)*, number 3321 in LNCS, pages 93–108. Springer-Verlag, December 2004.

[CKW93] W. Chen, M. Kifer, and D.S. Warren. HiLog: A foundation for higher order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[CMM+06] M. Carro, J. Morales, H.L. Muller, G. Puebla, and M. Hermenegildo. High-Level Languages for Small Devices: A Case Study. In Krisztian Flautner and Taewhan Kim, editors, *Compilers, Architecture, and Synthesis for Embedded Systems*, pages 271–281. ACM Press / Sheridan, October 2006.

[Con83] 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.

[Con87] J. S. Conery. *Parallel Execution of Logic Programs*. Kluwer Academic Publishers, Norwell, Ma 02061, 1987.

[CW94] M. Carlsson and J. Widen. *Sicstus Prolog User's Manual*. Po Box 1263, S-16313 Spanga, Sweden, April 1994.

[DEDC96] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, 1996.

[DeG84] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.

[DeG87] D. DeGroot. A Technique for Compiling Execution Graph Expressions for Restricted AND-parallelism in Logic Programs. In *Int'l Supercomputing Conference*, pages 80–89, Athens, 1987. Springer Verlag.

[dGCH+08] P. Chico de Guzmán, M. Carro, M. Hermenegildo, Claudio Silva, and Ricardo Rocha. An Improved Continuation Call-Based Implementation of Tabling. In D.S. Warren and P. Hudak, editors, *10th International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, volume 4902 of *LNCS*, pages 198–213. Springer-Verlag, January 2008.

[DL91] S. K. Debray and N.-W. Lin. Automatic complexity analysis for logic programs. In *Eighth International Conference on Logic Programming*, pages 599–613, Paris, France, June (1991). MIT Press.

[DL93] S. K. Debray and N. W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

[dlB94] M. García de la Banda. *Independence, Global Analysis, and Parallelism in Dynamically Scheduled Constraint Logic Programming*. PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, September 1994.

[dlBHM93] M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in Constraint Logic Programs. In *1993 International Logic Programming Symposium*, pages 130–146. MIT Press, Cambridge, MA, October 1993.

[dlBHM96] M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in dynamically scheduled logic languages. In *1996 International Conference on Algebraic and Logic Programming*, number 1139 in LNCS, pages 47–61. Springer-Verlag, September 1996.

[dlBHM00] M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in CLP Languages. *ACM Transactions on Programming Languages and Systems*, 22(2):269–339, March 2000.

[DLGH97] S.K. Debray, P. López-García, and M. Hermenegildo. Non-Failure Analysis for Logic Programs. In *1997 International Conference on Logic Programming*, pages 48–62, Cambridge, MA, June 1997. MIT Press, Cambridge, MA.

[DLGHL94] S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Estimating the Computational Cost of Logic Programs. In *Static Analysis*

Symposium, SAS'94, number 864 in LNCS, pages 255–265, Namur, Belgium, September 1994. Springer-Verlag.

[DLGHL97] S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[DLH90] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

[Fag87] B. S. Fagin. *A Parallel Execution Model for Prolog.* PhD thesis, The University of California at Berkeley, November 1987. Technical Report UCB/CSD 87/380.

[GHPSC94] G. Gupta, M. Hermenegildo, E. Pontelli, and V. Santos-Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *International Conference on Logic Programming*, pages 93–110. MIT Press, June 1994.

[GPA+01] G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: a Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, July 2001.

[Hal85] R. H. Halstead. MultiLisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[Han] Michael Hanus et al. Curry: An Integrated Functional Logic Language. `http://www.informatik.uni-kiel.de/~mh/curry/report.html`.

[Hau90] B. Hausman. Handling speculative work in or-parallel prolog: Evaluation results. In *North American Conference on Logic Programming*, pages 721–736, Austin, TX, October 1990.

[Her86] M. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 25–40. Imperial College, Springer-Verlag, July 1986.

[Her87] M. Hermenegildo. Relating Goal Scheduling, Precedence, and Memory Management in AND-Parallel Execution of Logic Programs. In *Fourth*

International Conference on Logic Programming, pages 556–575. University of Melbourne, MIT Press, May 1987.

[Her00] M. Hermenegildo. Parallelizing Irregular and Pointer-Based Computations Automatically: Perspectives from Logic and Constraint Programming. *Parallel Computing*, 26(13–14):1685–1708, December 2000.

[HF00] S. Haridi and N. Franzén. *The Oz Tutorial*. DFKI, February 2000. Available from http://www.mozart-oz.org.

[HG91] M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[HLA94] L. Huelsbergen, J. R. Larus, and A. Aiken. Using Run-Time List Sizes to Guide Parallel Thread Creation. In *Proc. ACM Conf. on Lisp and Functional Programming*, June 1994.

[HN86] M. Hermenegildo and R. I. Nasr. Efficient Management of Backtracking in AND-parallelism. In *Third International Conference on Logic Programming*, number 225 in LNCS, pages 40–55. Imperial College, Springer-Verlag, July 1986.

[HPBG05] M. Hermenegildo, G. Puebla, F. Bueno, and P. López García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[HPBLG05] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[HR89] M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.

[HR90] M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.

[HR95] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.

[Hua85] C.H. Huang. An interpreter of restricted and-parallelism for prolog programs. MCC AI Note, 1985.

[Hue93] L. Huelsbergen. Dynamic Language Parallelization. Technical Report 1178, Computer Science Dept. Univ. of Wisconsin, September 1993.

[HW87] M. Hermenegildo and R. Warren. Designing a High-Performance Parallel Logic Programming System. *Computer Architecture News, Special Issue on Parallel Symbolic Programming*, 15(1):43–53, March 1987.

[Jan94] Sverker Janson. *AKL. A Multiparadigm Programming Language*. PhD thesis, Uppsala University, 1994.

[JL89] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.

[JL92] D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent And-Parallelism. *Journal of Logic Programming*, 13(2 and 3):291–314, July 1992.

[Kal87a] L. V. Kalé. Completeness and Full Parallelism of Parallel Logic Programming Schemes. In *Fourth IEEE Symposium on Logic Programming*, pages 125–133. IEEE, 1987.

[Kal87b] L. V. Kalé. Parallel Execution of Logic Programs: the REDUCE-OR Process Model. In *Fourth International Conference on Logic Programming*, pages 616–632. Melbourne, Australia, MIT Press, May 1987.

[Kap88] S. Kaplan. Algorithmic Complexity of Logic Programs. In *Logic Programming, Proc. Fifth International Conference and Symposium, (Seattle, Washington)*, pages 780–793, 1988.

[KB88] A.H. Karp and R.C. Babb. A Comparison of 12 Parallel Fortran Dialects. *IEEE Software*, September 1988.

[KL88] B. Kruatrachue and T. Lewis. Grain Size Determination for Parallel Processing. *IEEE Software*, January 1988.

[LGBH05] P. López-García, F. Bueno, and M. Hermenegildo. Determinacy Analysis for Logic Programs Using Mode and Type Information. In *Proceedings of the 14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, number 3573 in LNCS, pages 19–35. Springer-Verlag, August 2005.

[LGHD96] P. López-García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 21(4–6):715–734, 1996.

[LH85] G. J. Lipovski and M. Hermenegildo. B-LOG: A Branch and Bound Methodology for the Parallel Execution of Logic Programs. In *1985 IEEE International Conference on Parallel Processing*, pages 560–568. IEEE Computer Society, August 1985.

[Lin88] Y.-J. Lin. *A Parallel Implementation of Logic Programs*. PhD thesis, Dept. of Computer Science, University of Texas at Austin, Austin, Texas 78712, August 1988.

[LK88] Y. J. Lin and V. Kumar. AND-Parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results. In *Fifth International Conference and Symposium on Logic Programming*, pages 1123–1141. MIT Press, August 1988.

[Lus88] E. Lusk et al. The Aurora Or-Parallel Prolog System. In *International Conference on Fifth Generation Computer Systems*. Tokyo, November 1988.

[Lus90] E. Lusk et al. The Aurora Or-Parallel Prolog System. *New Generation Computing*, 7(2,3):243–271, 1990.

[MBdlBH99] K. Muthukumar, F. Bueno, M. García de la Banda, and M. Hermenegildo. Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. *Journal of Logic Programming*, 38(2):165–218, February 1999.

[MCH04] J. Morales, M. Carro, and M. Hermenegildo. Improving the Compilation of Prolog to C Using Moded Types and Determinism Information. In *Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages*, number 3057 in LNCS, pages 86–103, Heidelberg, Germany, June 2004. Springer-Verlag.

[MCH07] J.F. Morales, M. Carro, and M. Hermenegildo. Towards Description and Optimization of Abstract Machines in an Extension of Prolog. In Germán Puebla, editor, *Logic-Based Program Synthesis and Transformation (LOPSTR'06)*, number 4407 in LNCS, pages 77–93, July 2007.

[MCN08] P. Moura, P. Crocker, and P. Nunes. High-level multi-threading programming in logtalk. In D.S. Warren and P. Hudak, editors, *10th International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, volume 4902 of *LNCS*, pages 265–281. Springer-Verlag, January 2008.

[MdlBH94] K. Marriott, M. García de la Banda, and M. Hermenegildo. Analyzing Logic Programs with Dynamic Scheduling. In *20th. Annual ACM Conf. on Principles of Programming Languages*, pages 240–254. ACM, January 1994.

[MG89] C. McGreary and H. Gill. Automatic Determination of Grain Size for Efficient Parallel Processing. *Communications of the ACM*, 32, 1989.

[MH89] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press, October 1989.

[MH90] K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *Int'l. Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

[MH92] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

[MLGCH08] E. Mera, P. López-García, M. Carro, and M. Hermenegildo. Towards Execution Time Estimation in Abstract Machine-Based Languages. In *10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 174–184. ACM Press, July 2008.

[MLGP+07] E. Mera, P. López-García, G. Puebla, M. Carro, and M. Hermenegildo. Combining Static Analysis and Profiling for Estimating Execution Times. In *Ninth International Symposium on Practical Aspects of Declarative Languages*, number 4354 in LNCS, pages 140–154. Springer-Verlag, January 2007.

[MNRA89] J.J. Moreno Navarro and M. Rodríguez-Artalejo. BABEL: A functional and logic programming language based on constructor discipline and narrowing. In *Conf. on Algebraic and Logic Programming (ALP)*, LNCS 343, pages 223–232, 1989.

[Nai91] Lee Naish. Adding equations to NU-Prolog. In *Proceedings of The Third International Symposium on Programming Language Implementation and Logic Programming (PLILP'91)*, number 528 in Lecture Notes in Computer Science, pages 15–26, Passau, Germany, August 1991. Springer-Verlag.

[Nar90] S. Narain. Lazy evaluation in logic programming. In *Proc. 1990 Int. Conference on Computer Languages*, pages 218–227, 1990.

[NM88] G. Nadathur and D. Miller. An overview of λprolog. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pages 810–827. MIT Press, 1988.

[NMLGH07] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *International Conference on Logic Programming (ICLP)*, volume 4670 of *LNCS*, pages 348–363. Springer-Verlag, September 2007.

[O'K90] R.A. O'Keefe. *The Craft of Prolog*. MIT Press, 1990.

[PBH00] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[PG98] E. Pontelli and G. Gupta. Efficient Backtracking in And-Parallel Implementations of Non-Deterministic Languages. In T. Lai, editor, *Proc. of the International Conference on Parallel Processing*, pages 338–345. IEEE Computer Society, Los Alamitos, CA, 1998.

[PGH95] E. Pontelli, G. Gupta, and M. Hermenegildo. &ACE: A High-Performance Parallel Prolog System. In *International Parallel Processing Symposium*, pages 564–572. IEEE Computer Society Technical Committee on Parallel Processing, IEEE Computer Society, April 1995.

[PGT+96] E. Pontelli, G. Gupta, D. Tang, M. Carro, and M. Hermenegildo. Improving the Efficiency of Nondeterministic And–parallel Systems. *The Computer Languages Journal*, 22(2/3):115–142, July 1996.

[PK88] V. Penner and A. Klinger. AND-Parallel PROLOG on Large Scale Transputer-Systems. Internal report, Institute for Applied Mathematics at the Aachen Technical University, West Germany, 1988.

[Qui86] Quintus Computer Systems Inc., Mountain View CA 94041. *Quintus Prolog User's Guide and Reference Manual—Version 6*, April 1986.

[San93] Vítor Manuel de Morais Santos-Costa. *Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I.* PhD thesis, University of Bristol, August 1993.

[Sar90] V. Sarkar. Instruction Reordering for Fork-Join Parallelism. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 25, pages 322–336, June 1990.

[SC99] Vítor Santos-Costa. Optimising Bytecode Emulation for Prolog. In *International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, volume 1702 of *LNCS*, pages 261–277. Springer-Verlag, 1999.

[SCWY91] V. Santos-Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–93. ACM, April 1991. SIGPLAN Notices vol 26(7), July 1991.

[SH96] K. Shen and M. Hermenegildo. Flexible Scheduling for Non-Deterministic, And-parallel Execution of Logic Programs. In *Proceedings of EuroPar'96*, number 1124 in LNCS, pages 635–640. Springer-Verlag, August 1996.

[SHC96] Z. Somogyi, F. Henderson, and T. Conway. The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language. *JLP*, 29(1–3), October 1996.

[She96] K. Shen. Overview of DASWAM: Exploitation of Dependent And-parallelism. *Journal of Logic Programming*, 29(1–3):245–293, November 1996.

[Sze89] P. Szeredi. Performance Analysis of the Aurora Or-Parallel Prolog System. In E. Lusk and R. Overbeek, editors, *North American Conference on Logic Programming*, pages 713–732. MIT Press, October 1989.

[Van94] P. Van Roy. 1983-1993: The Wonder Years of Sequential Prolog Implementation. *Journal of Logic Programming*, 19/20:385–441, 1994.

[War83] D.H.D. Warren. An Abstract Prolog Instruction Set. TR 309, SRI International, 1983.

[War87a] D.H.D. Warren. OR-Parallel Execution Models of Prolog. In *Proceedings of TAPSOFT '87*, Lecture Notes in Computer Science. Springer-Verlag, March 1987.

[War87b] D.H.D. Warren. The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation. In *International Symposium on Logic Programming*, pages 92–102. San Francisco, IEEE Computer Society, August 1987.

[WR87] H. Westphal and P. Robert. The PEPSys Model: Combining Backtracking, AND- and OR- Parallelism. In *International Symposium on Logic Programming*, pages 436–448. San Francisco, IEEE Computer Society, August 1987.

[WW88] W. Winsborough and A. Waern. Transparent And-Parallelism in the Presence of Shared Free variables. In *Fifth International Conference and Symposium on Logic Programming*, pages 749–764, Seattle,Washington, 1988.

[ZTD+92] X. Zhong, E. Tick, S. Duvvuru, L. Hansen, A.V.S. Sastry, and R. Sundararajan. Towards an Efficient Compile-Time Granularity Analysis Algorithm. In *Proc. of the 1992 International Conference on Fifth Generation Computer Systems*, pages 809–816. Institute for New Generation Computer Technology (ICOT), June 1992.

Vita

Amadeo Casas Cuadrado was born in Valladolid, Spain, on January 23, 1980, the son of Domitilo Casas Martínez and Guadalupe Cuadrado Ordax. After completing his studies at the Pinar de la Rubia School, Valladolid, Spain, in 1997, and finishing the eighth grade of piano studies with Honors at the Valladolid Conservatory of Music, he entered the Computer Science Department at the University of Valladolid. He was employed by Telefónica R&D from 2002 to 2003, while finishing his studies. He received his Bachelor's Degree in Computer Science in July 2003. In September 2003, he was awarded a Prince of Asturias graduate research fellowship in Information Science and Technology at the University of New Mexico. He entered the Electrical and Computer Engineering Department (ECE) in January 2004, and received his Master's Degree in Computer Engineering in December 2005. He was then admitted to candidacy for the Doctoral Degree and has been continuing his graduate work in the ECE Department of this university. During this time, Amadeo Casas Cuadrado has received his Master of Business and Administration from Robert O. Anderson School of Management, and has been active in serving the community in the state of New Mexico.