

Manuel Eduardo Carvalho Duarte Correia

On the Implementation of And/Or Parallel Logic Programming Systems



Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2001

Manuel Eduardo Carvalho Duarte Correia

On the Implementation of And/Or Parallel Logic Programming Systems



*Tese submetida à Faculdade de Ciências da
Universidade do Porto para obtenção do grau de Doutor
em Ciência de Computadores*

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto

2001

To Zica, with Love.

Acknowledgments

It is impossible to reach to the end of a project of this size without the influence, guidance and the indispensable support of innumerable individuals.

I would like to start to address my gratitude to Enrico Pontelli and Gopal Gupta for their patience and availability while I visited New Mexico State University a few years ago. Their work in this area has been and continues to be a great source of inspiration. I would also like to thank Kish Shen and the help he gave during the visit he made to Oporto University.

I dedicate this work:

To my colleagues at the Computer Science Department to whom I present my most sincere thanks for the companionship and support they have given me, in particular to José Paulo Leal, my business associate in other endeavours; to António Mário Florido, old companion and conspirator in the “malmequer” project; to Luís Lopes for the fantastic wild humour (we have to make a TV Show, someday) and, of course, to Sabine Broda, Ana Paula Tomás, Pedro Medas, Sandra Alves and Pedro Vasconcelos for the companionship and good moments spent together. I would also like to take the opportunity to express my gratitude to Miguel Filgueiras for the help he gave with the French version of the abstract.

To my office companions, Ricardo Rocha, Ricardo Lopes and Michel Ferreira to whom I would like to thank for the good working environment they have helped to develop at office 213. They have also played an important role in amassing our huge collection of .mp3 files. Without the innumerable hours of good music thus provided I have no doubt that it would have been much more difficult to finish this thesis in time.

To my supervisors. Fernando Silva and Vítor Santos Costa. To Fernando I must thank the continuous and unfaltering support he gave me during the long years it took me to finish. To both, because of the immense comprehension and patience they demonstrated towards my ever distracting extra activities, without which I most certainly would not be writing this lines. To Vítor I must thank the ultra-productive brain storms that formed the basis for this work. He is also responsible for turning my initial written gibberish into something readable. To Fernando I must thank the attention given to the thesis structure and to the little bugs that always find a way to creep into the n^{th} revision of the text. All remaining errors are therefore of my sole responsibility. To both I would also like to thank the opportunity they gave me to

develop and become the person I am today. I Thank You.

On a more personal note I would also like to dedicate this work to my family, without whom I would never had found the emotional stability necessary to finish.

To my parents, for always believing in me. To my brother Pedro, for not forgetting to always ask me what happened to the thesis. To my wife, Zica, for being there always at my side. To my son Manuel Luís for making all my days worthwhile and to my still unborn child, Eduardo Miguel, for making my life doubly happy. I Love You.

Porto, August 31, 2001

Manuel Eduardo C. D. Correia

Abstract

One of the advantages of logic programming is that it offers several sources of *implicit* parallelism, such as and-parallelism (ANDP) and or-parallelism (ORP). Recent research has concentrated on integrating the different forms of parallelism into a single combined system. In this work we deal with the problem of integrating ORP and independent and-parallelism (IAP), the two forms of parallelism most suitable for parallel Prolog systems. We contend that previous data structures do not allow for orthogonality between ANDP and ORP, leading to overly complex and inefficient designs. Our first major contribution thus includes the first complete analysis of the problem, from which we derive **(i)** a novel data-structure, the SBA (*Sparse Binding Array*), that does guarantee *memory independence* between the two forms of a parallelism thus leading to a more orthogonal integration, and **(ii)** detailed mechanisms for the efficient combination of IAP and ORP.

We then proceed to apply these mechanisms in the integration of a RAP-WAM derived IAP system with a SRI based ORP system. Several difficulties found in the process lead us to the conclusion that traditional designs for the implementation of IAP are more complex than what one needs, thus difficulting integration with or-work. As a case in point, backtracking in the presence of IAP is very complex and will interfere badly with existing or-parallel systems.

Our second major contribution is a novel model for IAP execution, fully supporting backtracking and memory reuse. This new model, LGSATS, learns from experience in previous ORP and IAP systems and is arguably much more amenable to be integrated with a SBA based ORP system. Our novel design is based on the following main guidelines: **(i)** And-agents perform as sequential engines for the most part of their execution; **(ii)** we follow a lazy work-publishing strategy in order to induce large grain and-tasks for parallel execution; **(iii)** we re-utilize WAM data-structures wherever possible; **(iv)** we initialize and-parallel data structures only when they are publicized. Our contributions simplify system implementation, and facilitate integration between the two forms of parallelism.

Resumo

Uma das vantagens da programação em Lógica é permitir a exploração *implícita* de várias formas de paralelismo, nomeadamente paralelismo ou- e paralelismo e-. Mais recentemente, a investigação na área concentrou-se em integrar as diferentes formas de paralelismo num único sistema combinado. Este trabalho trata do problema de integrar paralelismo e- independente (IAP) com paralelismo ou- (ORP), que são reconhecidamente as duas formas de paralelismo mais adequadas para sistemas Prolog paralelos. Na tese argumenta-se que as estruturas de dados que têm vindo a ser propostas para integração não permitem ortogonalidade entre o paralelismo ou- e o paralelismo e-, do que resultam sistemas demasiado complexos e ineficientes. A nossa primeira contribuição inclui uma análise completa do problema de onde depois se deriva **(i)** uma nova estrutura de dados, a SBA (*Sparse Binding Array*), que garante a independência em termos de memória entre as duas formas de paralelismo, tornando a integração mais ortogonal, e **(ii)** mecanismos detalhados para uma combinação eficiente de IAP com ORP.

Procedemos depois à aplicação destes mecanismos na integração de um sistema e- independente, a RAP-WAM, com um sistema ou- paralelo SRI. As várias dificuldades encontradas levaram-nos a concluir que os mecanismos tradicionais que têm vindo a ser usados para a implementação de IAP são demasiado complexos, o que dificulta a integração com trabalho ou-. Como ilustração basta referir que o processo de ‘backtracking’ na presença de IAP é muito complexo e integra-se mal nos sistemas actuais ou- paralelos.

A nossa segunda contribuição é portanto um novo modelo para a execução IAP, que suporta ‘backtracking’ e reutilização de memória. Este novo modelo, LGSATS, tem em linha de conta a experiência entretanto ganha na integração de paralelismo ou- com paralelismo e- independente, sendo mais fácil de integrar num sistema ou- paralelo baseado na SBA. Este novo sistema baseia-se nos seguintes princípios: **(i)** os agentes e- funcionam como motores sequenciais durante a maior parte do seu período de

execução; **(ii)** o sistema segue uma estratégia “preguiçosa” de publicação de trabalho e- de modo a induzir tarefas e- paralelas com elevada granularidade; **(iii)** as estruturas de dados da WAM são reutilizadas sempre que possível, **(iv)** as estruturas de dados paralelas só são inicializadas quando tornadas públicas. As nossas contribuições simplificam a implementação do sistema e facilitam a integração entre estas duas formas de paralelismo.

Résumé

Une des avantages de la Programmation Logique c'est que elle offre plusieurs sources de parallélisme *implicite*, telles que ANDP et ORP. Récemment, la recherche s'est concentré sur la intégration de ces différentes formes de parallélisme dans un seule système combiné. Dans ce travail nous étudions le problème de la intégration ORP et IAP, les deux formes du parallélisme les plus appropriées aux systèmes Prolog parallèles. Notre conclusion est que les structures de données précédentes ne tiennent pas compte de l'orthogonalité entre ANDP et ORP, menant excessivement à des conceptions complexes et inefficaces. Notre première contribution inclut la première analyse complète du problème, d'où nous dérivons **(i)** une nouvelle structures de données, le SBA, qui garantit l'indépendance de memoire entre les deux formes de parallélisme menant de ce fait à une intégration plus orthogonale, et **(ii)** des mécanismes détaillés pour la combinaison efficace de ANDP et de ORP.

Nous procédons alors a appliquer ces mécanismes dans l'intégration d'un système IAP dérivé par RAP-WAM avec un système ORP basé par SRI. Plusieurs difficultés trouvées dans ce processus nous mènent à la conclusion que les conceptions traditionnelles pour la mise en place de IAP sont plus complexes que de ce qu'on a besoin, de ce fait ont difficile l'intégration avec le travaille ou-. À titre d'exemple, le retour arrière en présence de IAP est très complexe et combine mal avec les systèmes parallèles-ou existants.

Notre deuxième contribution principale est ainsi un nouveaux modèle pour l'exécution IAP, avec retour arrière et de la réutilisation complète de la mémoire. Ce nouveau modèle, LGSATS, apprend de l'expérience des systèmes parallèles-et indépendants et parallèles-ou précédents et est discutablement beaucoup plus favorable pour être intégré avec un système parallèle-ou basé par SBA. Notre conception est basée sur les directives principales suivantes: **(i)** les Agents-et exécutent comme les moteurs séquentiels pour la plupart de leur exécution; **(ii)** nous suivons une stratégie d'édition de travail "paresseux" afin d'induire tâches-et de grain suffisamment large pour

l'exécution parallèle; **(iii)** nous réutilisons les structures de données de la WAM dans la mesure du possible; **(iv)** les structures de données parallèles-et sont initialisés seulement quand elles sont rendues publiques. Nos contributions simplifient la mise en place du système, et facilitent l'intégration entre les deux formes du parallélisme.

Contents

Abstract	5
Resumo	7
Résumé	9
List of Tables	19
List of Figures	22
1 Introduction	23
1.1 The Emergence of the Parallel Computer	24
1.2 Expressing Parallelism	25
1.3 The Case for Declarative Languages	27
1.3.1 Implicit Parallelism	28
1.3.2 Prolog as a Target Language	29
1.4 Parallel Machine Architectures and Prolog	30
1.5 Research Objectives	31
1.5.1 Design Goals	31
1.6 Main Contributions	32
1.7 Thesis outline	33

2	Implementation Technology and Parallelism	37
2.1	Logic Programming	37
2.2	Prolog – The language	40
2.2.1	Prolog History	42
2.3	Sequential implementation of Prolog	43
2.3.1	The Beginnings	43
2.3.2	The WAM	43
2.4	Other Logic Programming Languages	47
2.5	Prolog as a parallel programming language.	48
2.6	Implicit Parallelism in Logic Programs	50
2.7	Or-Parallelism	50
2.7.1	The Multi-sequential Model	51
2.7.2	Environment Copying	52
2.7.3	Binding arrays for or-parallelism	53
2.7.4	Hash Tables	54
2.7.5	Or-scheduling	55
2.8	And-Parallelism	56
2.9	Independent-And Parallelism	57
2.9.1	Extending the WAM for IAP	58
2.9.2	Backward Execution with CGEs	60
2.9.3	&-Prolog	61
2.10	The Committed-Choice Languages	62
2.11	Andorra Based Systems	63
2.12	Summary	65
3	Combined And/Or systems	67

3.1	Execution models for And/Or parallelism	69
3.1.1	Reuse-Based Execution Models	69
3.1.2	Recomputation-based execution models.	69
3.2	Binding Arrays for And/Or Parallelism	71
3.2.1	The Paged Binding Array	73
3.3	Copying for And/Or Parallelism.	75
3.3.1	Copy on write.	77
3.4	The Fire Model	79
3.5	DAOS – Scalable And/Or Parallelism	80
3.6	And/Or speculative work	82
3.7	Summary	84
4	SBA and the Orthogonality Principle	87
4.1	Orthogonality.	87
4.1.1	Teams and Epochs.	88
4.2	Sharing Work in the C-Tree.	90
4.3	Memory Independence	92
4.4	Sharing in the C-Tree.	93
4.4.1	Memory Independence in the PBA	93
4.4.2	Memory Independence in ACE.	94
4.4.3	Deterministically-Pure Reuse.	95
4.4.4	Detecting the source of conflict	97
4.4.5	Towards Memory Independence	97
4.5	The Sparse Binding Array	98
4.5.1	Comparing the SBA with other systems	100
4.5.2	System objects private values	101

4.6	Summary.	102
5	Or-Parallelism and the SBA	103
5.1	SBA Memory Allocation	103
5.2	The Binding Array in Aurora	104
5.3	The SBA in Aurora	105
5.4	Age and variable representation in the SBA	106
5.5	The SBA engine in the Private Region	109
5.6	Parallel Work	110
5.7	Initial Performance Evaluation	110
5.8	Performance comparison with other systems	111
5.9	Summary	114
6	Integrating IAP/ORP with the SBA	117
6.1	Or-workers as teams of and-agents	117
6.2	The SBA team process model	118
6.2.1	Migrating And-Agents	118
6.3	And-Agents versus Or-Parallelism	119
6.4	Or-Sharing And-parallel Data Structures	121
6.4.1	Goal Computation slots	122
6.5	The Team Or-Representative	123
6.6	Generalising the P-WAM towards IAP/ORP	124
6.6.1	Extra Machine Registers	125
6.6.2	Base Fields	126
6.6.3	Computation slots	126
6.6.4	End Markers	128

6.6.5	Parcall Frames	128
6.6.6	The Goal List	129
6.7	LPCO in And/Or Systems	130
6.8	Summary	131
7	Independent And-Parallelism Revisited.	133
7.1	Design Guidelines	134
7.2	Principles of Forward Execution	136
7.2.1	Memory Management	137
7.2.2	Control Flow in Forward Execution	138
7.3	Data-Structures for LGSATS Execution	141
7.3.1	Goal-frames and and-agent run-queues	141
7.3.2	Parallel Environments.	142
7.3.3	And-Task Nodes.	145
7.4	New Machine Registers	148
7.5	Compiling forward execution.	149
7.5.1	CGE with no sequential goals to the right.	150
7.5.2	CGE with Sequential Goals In the Continuation.	154
7.5.3	CGE with Sequential Goals to the Left.	154
7.6	LGSATS Overheads	157
7.7	And/Or Tree IAP-Levels	159
7.7.1	Moving In The And/Or-Tree	160
7.7.2	Maintaining the <i>Minimum Reached Level</i>	162
7.8	Searching for And-Work in the And/Or Tree.	163
7.9	Publishing And-Work.	164
7.10	Memory Organization in Detail.	166

7.11 Summary.	169
8 Backtracking under IAP Revisited.	171
8.1 Backtracking in LGSATS.	171
8.1.1 Private Backtracking	172
8.1.2 Public Backtracking Within the Same And-Task	173
8.1.3 Public Backtracking To A Different Task	175
8.2 The Backtracking-Frame	177
8.3 Backtracking Messages	178
8.3.1 Reachability	178
8.4 Piranha Backtracking Mode.	180
8.4.1 Undoing Incomplete Parallel-Environments	181
8.4.2 Untrailing	182
8.4.3 Propagating Failure	182
8.5 Feeding Frenzy Backtracking Mode.	184
8.6 The Decision Mode.	184
8.7 Transitive Backtracking	186
8.8 Inside, Outside and Regular Backtracking.	187
8.8.1 Inside Backtracking.	188
8.8.2 Outside Backtracking.	189
8.8.3 Regular Backtracking.	189
8.8.4 Iap-Levels After Successful Backtracking.	190
8.9 An Example of Inside-Backtracking.	190
8.10 An Example of Regular-Backtracking.	193
8.11 An Example of Outside-Backtracking.	193
8.12 Memory Reuse by Reference Counting.	196

8.13 Summary.	199
9 Conclusions and further work	201
9.1 Contributions	201
9.1.1 The Orthogonality Principle	202
9.1.2 Memory Independence	202
9.1.3 The Sparse Binding Array	203
9.1.4 LGSATS	203
9.2 Further Work	204
9.3 Final Remarks	205
References	207

List of Tables

2.1	WAM code for Naive Reverse	46
5.1	Performance of SBA on a SparcCenter 2000	112
5.2	Performance of Aurora on a SparcCenter 2000	112
5.3	Overheads Yap Prolog/Or-Parallel Models with one worker.	113
5.4	Speedups for the three models on the PC Server.	114
5.5	Speedups for the three models on the SparcCenter.	114

List of Figures

2.1	WAM data areas and registers	45
2.2	Binding Arrays in or-parallelism.	54
2.3	RAP-WAM data areas and registers	58
3.1	Goal c being recomputed thrice.	70
3.2	Using the binding array in and-parallelism.	72
3.3	The Paged Binding Array.	74
3.4	Daos Worker Organization.	82
3.5	or-and speculative work.	84
4.1	Epoch-Based And/Or Parallelism.	90
4.2	Inside killing of a public or-branch.	91
4.3	Structural problem in PBA.	93
4.4	Structural problem in ACE.	95
4.5	Deterministically-pure reuse policy.	96
4.6	The Sparse binding array.	99
5.1	The binding array in Aurora.	105
5.2	Conditional bindings under a parcall.	106
6.1	And-Agent Lifecycle.	120

6.2	Or-public and or-private goals.	122
6.3	Data structures for or- and iap-and Parallelism.	125
6.4	Computation Slot states.	127
7.1	A CGE-environment in LGSATS.	142
7.2	Code for CGE With No Sequential Continuation	151
7.3	Code for CGE With Sequential Goals In the Continuation	155
7.4	Illustrating the need for the <code>start_cge</code> LGSATS instruction.	156
7.5	Code for CGE With Sequential Goals to the Left.	157
7.6	Moving up the and/or tree within the same sequential task.	161
7.7	Deterministic continuation of a parallel environment.	163
7.8	Parallelizing cge-environments in LGSATS.	165
7.9	Tasks memory organization.	168
8.1	And-private IAP backtracking.	172
8.2	And-public IAP backtracking within the same and-task.	174
8.3	IAP external backtracking.	176
8.4	Detecting an INSIDE and REGULAR backtracking operations.	191
8.5	Detecting an OUTSIDE backtracking operation.	194
8.6	Reference counting for memory management.	197

Chapter 1

Introduction

Parallelism allows several computers to solve the same problem together. Original expectations were for cheap parallel super-computers that would be capable of solving a wide variety of challenging problems at a very low price. Unfortunately, these expectations have not yet been fulfilled. This situation is now changing. Gigabit networks are becoming commonplace in LANs and the price/performance ratio of “off-the-shelf” PC-components is still decreasing. As a result, it is nowadays possible and inexpensive to build parallel computers using PC-components and gigabit networking and thus obtain parallel machines with a very good price/performance ratio [134, 78]. At last, progress in technology may allow the full emergence of the parallel computer as an inexpensive path to increase performance.

In this thesis we address one major problem that has further delayed the progress of parallelism, namely, how to effectively program parallel computers. We approach this problem from the view-point of logic programming, a high-level approach to programming that seems particularly amenable to parallel execution. It has been shown that Prolog programs can benefit from parallelism, and moreover, that parallelism in these programs can be exploited with little programmer intervention. This is because Prolog programs exhibit several forms of parallelism. Depending on the application, one may be dominant or several may exist at the same time.

Unfortunately, previous work on parallel Prolog only allowed for the exploitation of a single form of parallelism in a single system. In this thesis, we present an unified framework that allows for the simultaneous exploitation of the several forms of parallelism that can be found in Prolog programs.

1.1 The Emergence of the Parallel Computer

Parallelism is about several CPUs working together to solve the same problem. This is not a new idea. Computer pioneers thought of ever faster computing machines created by harnessing the power of multiple processing elements. However parallel computers turned out to be more difficult to deploy and use within mainstream computing than originally thought. In the first three decades (1950-1980) the design and construction of the parallel hardware was extremely challenging. At that time parallel machines were highly unreliable and the hardware costs were so high that everyone simply ignored the unsolved problems of software. Just to make these machines work for a couple of hours proved to be challenging enough.

In the middle 1980s hardware became more reliable and inexpensive, thus making the production of parallel computers feasible. At about the same time microprocessor-based parallel machines started to appear and several small companies sprung into existence to exploit the new emergent market of microprocessor based parallel computers ¹.

At the time, and mostly because of several integration problems, mainstream microprocessors would lack builtin support for parallelism. Special circuitry, developed in house, would be required. In order to support development costs manufacturers had to sell these machines at a high prices, resulting in low volume sales. Market for these machines never expanded, eventually leading to the demise of some of these early companies.

Growing on-chip support for multiprocessor memory coherency in microprocessors has changed this picture. This support has allowed integration in mainstream technology, resulting in lower price and earlier deployment. We can now watch substantial market growth for parallel machines.

Nowadays, parallel computers play a major role in scientific computing in physics, chemistry, material sciences, biology, astronomy, earth sciences and others. These fields rely on computers to model physical phenomena. Such simulations are by now essential to many industries and would not be viable without parallelism. As a different application, modern large-iron database servers rely on parallel architectures. In fact, mainframes have used parallel technology for a very long time.

It is very unlikely that single-processor performance will continue to increase indefi-

¹Sequent, Encore, Flex, Synapse and Myrias, just to name a few.

nately, if for no other reason than physics. Aside from the limitations of packaging, heat dissipation, and switching speed of semiconductors, it seems plausible that fundamental laws of physics, such as speed of light and that circuit size will limit single processor performance. However, even before this happens, the only economical viable way to increase computational power will be to add more processors and make them work in parallel.

In [50] it is argued that a 30-fold improvement in microprocessor speed is still to be expected in little more than a decade (1993 to 2005) and is predicted that by 2013 the only way to attain significant performance gains in an economical way will be to add more processors. It is thus expected that the first decade of the twenty-first century will initiate “the age of parallelism”.

This trend is readily emerging. Machines with two CPUs are used as workstations, machines with four CPUs are being used as mid-size low-end servers and machines with eight and more processors are used as departmental servers and in supercomputer centers. This trend is bound to increase in the near future with more processors being used in workstations and low-end servers. We therefore believe that parallel computing will become even more significant and pervasive in the future.

1.2 Expressing Parallelism

One of the main problems in harnessing the full potential of parallel computers is how to effectively program them. No “silver bullet” has been found yet, however it is generally recognized that there are basically three main ways to tackle this problem

Implicit Parallelism — In this approach the runtime system discovers and exploits the parallelism present in a program. The programmer does not have to consider parallelism, as this is left to the compiler and/or runtime system. This scheme has the great advantage of giving the programmer more freedom to concentrate on the problem at hand instead of having to worry about parallelism. In this approach software development and testing is performed on a sequential system. Later on, the final code will run unchanged on a parallel machine. The single change the programmer should notice when his programs are run in parallel should be an improvement in terms of execution speed.

The complexity of achieving this goal with imperative languages is very high. Good results have been obtained for applications that manipulate arrays in

a repetitive fashion. Analysis has been found to be very complex for other applications [159], even in pointer-clean languages such as Fortran. In fact, parallel implementations of Fortran have to resort to user annotations such as the “do parallel” loop. Implicit exploitation of parallelism is however common and viable with declarative languages.

Declarative Parallelism — In this approach the programmer explicitly states where he *wants* parallelism to occur. The compiler and run-time system then take care of *how* to actually exploit parallelism in those points of the program. The indication of where parallelism is available within the program can be implicit in the language, as in the case of concurrent programming languages like PCN [53] or even concurrent logic programming based languages like Parlog [55]. Or it can be explicit as in the case of for example &-Prolog [74].

Operational Parallelism — In this case the programmer fully manages *where* and *how* the parallelism occurs in the program and how to exploit it. This is the most low-level approach, but it is also the one that gives the programmer more flexibility. In this case the programmer is generally provided with tools for dealing with parallelism at a certain level of abstraction. The tools go from specialized libraries supplying a uniform set of communications primitives that hide some details about the computing environment (e.g, PVM [135] and MPI [56]), to abstractions fully supported in the design of programming languages such as Ada [132] Occam [21] and SR [8].

Initial imperative parallel systems exploited *operational parallelism*. Modern imperative parallel systems (eg, Fortran90 or parallel-C languages) have moved or are moving towards *declarative parallelism*.

Logic programming based languages offer the ability to exploit *implicit parallelism* and *declarative parallelism* on the same framework. In contrast, low-level, truly *operational parallelism* paradigms are difficult and cumbersome to use within logic programming systems. This happens because in *operational parallelism* the program must interact closely with the real machine that is abstracted away in logic programming. We next discuss the origin and application of these languages.

1.3 The Case for Declarative Languages

The history of software development shows a trend towards making the task of programming easier, even at the cost of some efficiency. We thus have seen programming languages evolve from machine code, to assembler, to semi-high level languages such as Fortran, to structured languages like Pascal and C, and to declarative languages like Prolog, ML or Haskell.

Declarative languages allow the programmer to specify a problem at a more application-oriented level. In a nutshell, declarative languages allow the programmer to specify in the program what the problem is, instead of forcing him to address all the nitty-gritty details of how the problem should be solved by the computer [9].

Prolog is an example where a program is a set of axioms that describe in first order logic a set of objects in the problem domain and how they relate. Solving a problem is akin to formally prove a statement (or query) in first order logic against the model of the “world” described by the program. In Prolog, the proof is existential and returns an answer that is a set of conditions (or constraints) on the initial arguments that satisfy the original statement.

One criticism of declarative languages is that they perform badly at solving complex problems. An example are the numeric problems forming the “Grand Challenge Problems”² that require over Tflops of performance. At the moment, and in the near future, only parallel computers can tackle these problems. Parallel software for these problems is usually simpler as the problems tend to be very regular in nature, thus making current imperative programming languages adequate for these tasks.

Unfortunately, parallelism is no longer regular for other, more sophisticated applications. Examples abound in the artificial intelligence and symbolic computation field. Expert systems, natural language processing and computer aided design are examples of problems that are not as easy to parallelize and that indeed are difficult to tackle with conventional programming languages. Traditional approaches to parallel programming are proving to be inadequate for these complex applications. Too much time is dedicated to develop and debug parallelism, not to mention the human factor, as competent parallel programmers are hard to find and their time very expensive.

We thus argue for a programming language that would be sufficiently general to tackle

²Global Change, Human Genome, Fluid Turbulence, Vehicle Dynamics, Ocean Circulation, Superconductor Modeling, etc.

these problems whilst amenable to the implicit exploitation of parallelism. We next discuss declarative languages in general, and Prolog in particular, and study how well they are positioned to fill this gap.

The meaning of a declarative program is given by its *declarative semantics* obtained from an underlying formalism such as First-Order Logics or the Lambda-calculus. We are free to use several operational semantics, as long as they are consistent with the declarative meaning of the program. In contrast, the semantics of imperative programs must be “lifted” from an analysis of what a sequential execution would do, because this and only this is what the program specifies.

1.3.1 Implicit Parallelism

Declarative languages have a meaning that is independent of execution in a computer, therefore one can easily reason “mathematically” about a declarative program and perform “meaning” preserving transformations that extract or even control the implicit parallelism available in the program. This is very difficult to achieve in a non-declarative language.

To exploit implicit parallelism we can take advantage of:

Non-Determinism their operational semantics can be intrinsically non-deterministic, eg, clause selection in logic languages.

Dataflow style of computation that result from using eager evaluation schemes, such as unfolding in Reform Prolog [16] or parallel graph reduction in functional programming based languages.

However, in order to obtain efficient execution with the implicit exploitation of parallelism we must address several difficult problems. In a sense, the run-time system for declarative languages compiles implicit to explicit parallelism. It is therefore up to the language implementor to address all the complex details required to exploit parallelism. Among such problems:

- Good data-structures and algorithms for the exploitation of parallelism must be designed. Our design must be sufficiently flexible to perform well under very different programs. The problem is even harder if the same system must support several forms of parallelism.

- Task management must be done automatically. If not done carefully, we may have too few coarse grained tasks, that do not exploit all parallelism, or too many fine-grained tasks, that incur more costs than obtain benefits. In both cases we will have slower execution than what is achievable.
- The system may try to exploit parallelism in sections of code for which there is none. Parallelisation overhead often slows down execution.

Last, we note that in a few cases programmers may want to have control over parallelism, say, a programmer may feel he best knows which are the best sources for parallelism. This can be achieved by embedding in the language mechanisms for controlling the parts of the program that are allowed to run in parallel. Such a solution achieves the best of both worlds, as programmatic interfaces for parallelism control and implicit exploitation of parallelism are joined in a single system. As we shall see, parallel declarative languages in general, namely parallel Prolog systems, provide such an embedding.

1.3.2 Prolog as a Target Language

Various arguments can be made in favor of either functional or logic programming, and indeed each of the two paradigms have their detractors and defendants. Functional programming is often celebrated for its higher-order capabilities where functions are treated as “first-class citizens” of the language and for its ability to support a declarative view of I/O [17]. Logic programming on the other hand has non-determinism in its operational semantics and supports the powerful concept of *logic variables and unification*, thus allowing for bidirectionality of execution and for partial bindings.

We have chosen parallel logic programming, and Prolog in particular, as the main focus of our study because of its relevance for Artificial Intelligence style of applications, and because its non-deterministic nature and its logic based semantics are natural fits to the automatic extraction of parallelism. Regarding Prolog applications, we can name areas such as Expert Systems, Machine Learning, and Decision Systems where logic programming and in particular Prolog play an important role. Note that our results for Prolog also find direct application in Functional/Logic languages [67], a new combined paradigm.

Within logic programming, we choose Prolog as our target language because it is by far the most widely used logic programming language. It is also a language for which

a huge collection of “dusty-deck” applications exists just waiting to be automatically parallelized by a parallel Prolog run-time system.

There are two main sources of implicit parallelism in Prolog programs. *Or-parallelism* (ORP) arises from the parallel execution of multiple clauses capable of solving a goal, that is from exploring the non-determinism present in logic programs. *And-parallelism* (ANDP) arises from the parallel execution of multiple subgoals in a clause body. If the goals do not share, we say we are in the presence of *Independent And-Parallelism* (IAP).

1.4 Parallel Machine Architectures and Prolog

Multiprocessor machines may be classified as either *shared-memory* or *distributed memory* based machines. In a shared-memory based multiprocessor all processors have access to a single main memory. In a distributed memory machine each processor has its own private memory that is not directly accessible by the other processors. In the case of distributed memory based machines, synchronization among the different processors is achieved explicitly by message passing, while in shared-memory based machines synchronization can be achieved through low-level hardware supported synchronization mechanisms that take advantage of the shared memory nature of the machine. In this category we can include the *lock* of a memory position normally implemented as an atomic swap of the contents of a register and a shared memory position. This mechanism can then be used to implement higher level synchronization primitives such as *monitors* and *barriers* [18].

We envisage a shared-memory based system in this work. Note that it is possible to simulate through software a shared-memory machines in a distributed architecture. These systems utilize techniques similar to those used to maintain coherence on cache lines in hardware, but they implement the coherence algorithms in software by manipulate pages. Page or translation look-aside buffer (TLB) faults are used to trigger the coherence mechanisms. Early work on *virtual shared memory systems* was by Kai Li at Yale and Princeton Universities [86, 85]. Software implementations of these protocols are available for standard workstations connected by a LAN and also for message-passing multicomputers.

In our work we shall have no explicit control of task grain size. And-parallel tasks may be quite fine grained, therefore the exploitation of this form of parallelism in

distributed platforms requires very good granularity control. We therefore prefer to experiment with shared memory platforms first. Current work being developed for the DAOS [27] system and the distributed memory organization schemes embodied in COWL [111] give hints to how our platforms can be generalised to distributed platforms.

1.5 Research Objectives

The main objective for the research described in this thesis is to specify and derive in detail an efficient “machinery” for the simultaneous exploitation of And/Or parallelism in Prolog programs. Our motivation lies in good results on or-parallelism and independent and-parallelism, and on the lack of a sufficiently detailed specification which we could use to develop full implementation of an And/Or parallel system.

Our research started from proposals on the data-structures for And/Or-parallelism [64, 61]. These proposals were thought to allow efficient solution models for the transparent exploitation of And/Or parallelism in Prolog. Unfortunately, when experimented in the development of an actual implementation some subtle deficiencies started to emerge. We thus realized that a major redesign was required to make the And/Or integration of parallelism in Prolog a reality.

1.5.1 Design Goals

It is widely accepted and substantiated by simulation results [126] that both and- and or-parallelism do arise naturally in Prolog programs. An ideal Prolog system would be therefore capable of simultaneously exploit both forms of parallelism. Such a program should perform as well as or-parallel and and-parallel Prolog. Moreover, a better parallelisation and minimum runtime will be achieved in the cases where both forms of parallelism are present.

There are several design goals that we have tried to achieve from the beginning, while developing our combined And/Or parallel Prolog system. The system should :

- Support the implicit execution of Prolog programs without any intervention from the programmer.
- Support the simultaneous exploitation of both and- and or-parallelism.

- Match the performance of pure and-parallel system when only and-parallelism is being exploited.
- Match the performance of a pure or-parallel system when only or-parallelism is being exploited.
- Have one processor performance comparable to that of a state of the art sequential Prolog system.

These requirements will serve as our definition of what constitutes an efficient combined And/Or parallel Prolog system.

1.6 Main Contributions

We now resume what are the main contributions of these thesis:

- *The Orthogonality Principle* (section 4.1) — During our research it became clear that we should follow a major design principle, *orthogonality*: or-parallel execution should be unaware of independent and-parallel execution, and independent and-parallel execution should be able to ignore, as much as possible, the very existence of or-parallelism in the system.
- *Memory Independence* (section 4.3) — In studying a number of models described in the literature for combining And/Or parallelism (composition tree + stack copying or composition tree + paged binding array) [62, 59, 64] for full Prolog, we verified (section 4.3) that when we consider what happens within a team when it travels about the computation tree looking for work, memory allocated for shared ORP can conflict with the team internal use of memory for ANDP done in advance.

We say that teams are *memory independent* when the way memory is allocated within a team can *never restrict* the way another team is allowed to allocate memory. If memory allocation within a team depends on other teams we show that ANDP cannot be orthogonal to ORP.

- *The Sparse Binding Array* (section 4.5) — A major result of our work is a *new* environment representation scheme, the *Sparse Binding Array* (SBA). The SBA is an “orthogonal” and “memory independent” compliant approach for the

simultaneous exploitation of And/Or parallelism in Prolog programs. Chapter 5 presents experimental performance results that prove that the SBA is an efficient environment representation scheme. In Chapter 6 we give an overview of how IAP and ORP could interact in a SBA based combined system. We also explain how the data structures and execution models used in the implementation of binding arrays derived or-parallel systems (Aurora) and iap-parallel RAP-WAM [72, 75] derived systems (&-Prolog [74] and &ACE [105]) can be adapted to obtain a SBA recomputation based IAP/ORP execution model for full Prolog.

- *LGSATS* – In Chapter 7 we present a new IAP execution model, complete with backtracking and memory reuse, which incurs minimum overhead while executing non-public ANDP code. LGSATS is designed from the beginning to comply as much as possible with an orthogonal IAP/ORP integration.

The key ideas present in our design are: **(i)** to always take advantage of the analogy between ORP and IAP [103]; **(ii)** to avoid creating new structures by re-utilizing WAM data structures wherever possible; **(iii)** to avoid major changes to the compiler; **(iv)** to initialize and-parallel data structures only when they are publicized. We shall assume that the underlying system is WAM based [151], like YAP [42, 112]

The classical IAP backtracking mechanism is very complex. In Chapter 8 we take a fresh look at this issue and, by taking advantage of the new runtime execution behaviour provided by LGSATS and the notion of *iap-levels* (section 7.7) we come up with a simpler and hopefully more efficient way of performing IAP-backtracking operations.

1.7 Thesis outline

This dissertation contains 9 chapters:

Chapter 1 — We have tried to provide a general historical and conceptual background in parallel computing and understand the role declarative languages currently play in the field. We argue that declarative languages have a more important role to play in parallel systems, particularly through the implicit exploitation of parallelism. We then proceed by arguing the reason behind our choice of Prolog as the target language for our parallel system. We next delve into some considerations that lead us to justify our choice of target parallel machine architecture into which we base our system design.

We last present a general outline of the main aims and contribution of our work in the field.

Chapter 2 — We provide a general survey of parallel logic programming and associated implementation technology. After browsing through this chapter the reader should be able to acquire a sufficient background knowledge to fully appreciate and localize the importance and relevance of the present work in the field.

Chapter 3 — We study in more detail previous work on ANDP/ORP execution of Prolog from which our present work derives. We survey the main issues on combining ANDP with ORP, namely: reuse vs. recomputation, binding arrays based environment representations vs. copy based approaches and Hash Tables.

Chapter 4 — In this Chapter we describe some subtle problems we found with previous proposals when combining ANDP with ORP. We derive two general design principles, *Orthogonality* and *Memory Independence* that we use as guidelines to derive and propose a *new* environment representation data-structure, the Sparse Binding Array (SBA), that does not suffer from these difficulties whilst allowing for the natural integration of the main data-structures for both IAP and ORP.

Chapter 5 — We describe in detail some issues associated with the implementation of SBA on the Aurora system and benchmark the SBA derived system, concluding that the SBA approach is an efficient environment representation scheme for ORP. These results were very promising as they opened the way and showed the validity of using the SBA approach for the implementation of a combined IAP/ORP system.

Chapter 6 — In this chapter we discuss some of the modifications and extensions necessary for combining independent-and and or-parallelism. We conclude that updates to or-shared iap- structures can be treated as updates to multivalued or-shared variables. This is the insight that allows us to make extensive use of the SBA in the design of a simpler combined IAP/ORP Prolog system.

Chapter 7 — We come to terms with drawbacks in the classical approaches to the implementation of IAP that unnecessarily complicate IAP/ORP. We propose up with a new design for the integration of IAP and ORP that minimises overheads while in sequential mode of execution and tends to dynamically produce less but coarser grain-size and-tasks. We call this new scheme the *Large Grain Sequential And Tasks for SBA (LGSATS)*.

Chapter 8 — No implementation of Prolog, be it sequential or parallel is complete without a complete backtracking mechanism. In and-parallel systems this is generally

very complex and extremely difficult to implement in an efficient way. In this chapter we describe a *new* IAP backtracking scheme specially developed to take advantage of the new characteristics present in LGSATS. At the end, a mechanism based on reference counting is given for memory reuse upon backtracking.

Chapter 9 — This is the final chapter where we present some research conclusions and discuss future work that we think should orient part of the research that still needs to be done as a continuation of this work.

Chapter 2

Implementation Technology and Parallelism

In this chapter we survey work done in sequential and parallel implementations of logic programming derived systems. We start by giving an overview of terminology and basic concepts of logic programming, followed by a discussion on how they are applied in the Prolog language.

We then proceed by discussing the sequential implementation of Prolog since the original Marseille implementation. Focus is then given to the WAM, the basis for most Prolog systems. We then continue with parallel implementation issues. We present the main issues in parallel Prolog, followed by a discussion on the implementation of or-parallelism (ORP) and of and-parallelism (ANDP). Our attention last focuses into the issues related to independent and-parallelism (IAP).

2.1 Logic Programming

Logic programming [87] is a programming paradigm based in a subset of First order Logic [90] named Horn Clause Logic. A first order *clause* is a universally quantified disjunction of literals where a literal is a positive or negative atomic formula. A negative atomic formula is the logical negation of a positive atomic formula.

Each positive atomic formula A is of the form $p(t_1, \dots, t_n)$ where p is a predicate name and each of the t_i is a term. A term is either a *variable*, an *atom*, or a *compound term* of the form $f(t_1, \dots, t_n)$ where f is a functor of arity n and each of the t_i is a term.

Terms in a program represent “world objects” while predicates represent relationships among those objects.

A variable represents an unspecified term, that is, an unknown object. The process of *substituting* a variable for another term is known as *binding* the variable. A variable can be bound to another different variable or to a non-variable term. A term with no variables is a *ground term* and the process of binding variables in a term is known as *substitution*.

A horn clause is a clause where at most one of the literals is negative.

$$\sim A \vee B_1 \vee B_2 \dots \vee B_n \quad \text{or} \quad B_1 \vee B_2 \dots \vee B_n$$

This is equivalent to a logical implication of a conjunction of atomic formulae of the form:

$$A \Leftarrow B_1 \wedge B_2 \wedge \dots \wedge B_n \quad \text{or} \quad \text{false} \Leftarrow B_1 \wedge B_2 \wedge \dots \wedge B_n$$

That is normally simplified to the form:

$$A \leftarrow B_1, B_2, \dots, B_n \quad \text{or} \quad \leftarrow B_1, B_2, \dots, B_n$$

In all these representations the universal quantification is implicit. In the last form *false* is omitted if all the clause literals are positive. The literal *A* is called the head of the clause and the conjunction of the literals B_1, B_2, \dots, B_n is called the clause body. Literals in the body of a clause are also called subgoals. Clauses with an empty head are called *goals* or *queries*. Clauses consisting of just an *head* are called *facts*.

A logic program is a non-empty set of non query clauses. Given a program *P* and a query $B_1 \wedge B_2 \wedge \dots \wedge B_n$, program execution consists in finding the most general assignment Θ_{mgu} for the query variables such that $P \models \Theta_{mgu}(B_1 \wedge B_2 \wedge \dots \wedge B_n)$. As we shall see next, the proof is obtained by contradiction and is existential as it produces an assignment to variables present in the query.

The Prolog language relies on Robinson’s *Resolution* [109] with *Unification* as inference rule. Resolution allows for several possible strategies. One important strategy is linear resolution, where goals are always matched with a program’s clause. Because Horn Clauses have at most one positive literal, linear resolution is refutation com-

plete for these programs [90]. Prolog thus uses linear resolution, or more specifically SLD-Resolution (Select Linear Definite Resolution [87]).

SLD resolution proceeds in the following way. We want to prove that the negation of the original clause is inconsistent with the program. Given a query:

$$\leftarrow B_1, \dots, B_{i-1}, B_i, B_{i+1}, \dots, B_n$$

and a program P , it searches for a clause $H \leftarrow \overline{S}$ (\overline{S} is a set of positive literals) in the program such that B_i unifies with H with variable substitution Θ_{mgu}^j . B_i unifies with H iff $\Theta_{mgu}^j(B_i) =_{string} \Theta_{mgu}^j(H)$. The unification algorithm either produces the most general unifier (mgu) Θ_{mgu}^j of the two literals, or fails if no mgu can be found. The mgu is *unique*, module renaming of variables.

If unification succeeds, from resolution and from the initial query we obtain the resolvent clause:

$$\Theta_{mgu}^j(B_1, \dots, B_{i-1}, \overline{S}, B_{i+1}, \dots, B_n)$$

This resolvent is obtained by substituting the original query by a new query where B_i is substituted by the clause body and the substitution of variables computed by the unification of B_i with the clause head H is applied to the resulting query.

Notice that \overline{S} will be \emptyset if the program clause from which the resolvent is derived is a fact. If after a n -finite number of SLD-resolution steps, we end up with an empty resolvent the negation of the original query is inconsistent with the program, and thus we have proved that :

$$P \models \Theta_{mgu}^n(\Theta_{mgu}^{n-1}(\dots(\Theta_{mgu}^1(B_1, \dots, B_n))))$$

If failure occurs, the system *backtracks*, undoing successful resolutions, and tries to use an alternative clause. The process terminates either when it finds a solution or when all alternative clauses have been tried. This process can be described as exploring a tree known as the search-tree. The shape of this tree is determined by the *select_{literal}* rule. The internal nodes of this tree represent the successful unifications of a goal with the head of a clause. A leaf and its associated branch represents either a successful resolution (a proof) or a dead-end resulting from a failed unification.

SLD-resolution does not specify which literal in the query and which clause in the program should be selected in order to resolve a new query. In general, several pairs may be available, so SLD resolution is nondeterministic by nature.

It has been shown that the same final substitution will be made for the query independently of the *select_{literal}* rule we use, provided the execution terminates [87]. However, the amount of work, in terms of number of unifications can vary dramatically depending on this rule.

A computer implementation of resolution must specify *select_{literal}* and *select_{clause}*. Different languages and different semantics can thus be obtained. This was expressed by Kowalski with his now famous equation :

$$\textit{Algorithm} = \textit{Logic} + \textit{Control}$$

The equation means that the same program (the same logic) can implement different algorithms if a different *control* is used in evaluating the program. The interested reader can consult Kowalski's book [83] for further details.

Observe also that the nondeterminism implicit in the two selection rules for SLD-resolution can also be translated in *using parallelism* as a control rule in the Kowalski equation.

2.2 Prolog – The language

Prolog is a *logic programming* language where the *select_{literal}* rule is “for the resolvent to pick up the leftmost literal in the query” and *select_{clause}* follows the textual order of the program clauses. In the reminder of this thesis by “Prolog Semantics” we mean the operational semantics obtained by using this particular instantiation of the selection rules.

Prolog is a very flexible programming language and achieves its flexibility by extending pure *horn clause* logic with an extra collection of features that do not have an immediate logical counterpart. These features are indispensable in making Prolog a useful programming language for real world problems. They are :

Meta-logical predicates: These are used to manipulate terms in a non-logical way and to inquiry about the state of the computation (proof).

Side-Effect Predicates: Used to perform I/O and dynamically manipulate the set of clauses that constitute the program being executed.

Control Operators: The most well-known control operator is *cut*, which is used to dynamically trim the execution tree. When a programmer knows that no extra solutions can be obtained from other predicate clauses he can avoid those non useful computation branches by *pruning* away with the *cut* predicate. The cut prunes away alternative branches to the right (in the tree) from where the cut is encountered back to, and including, the head goal of the clause containing the cut. There are two kinds of cut, *green-cuts*, do not change the declarative reading of a Prolog program; programs with *red-cuts* can only be understood in the operational semantics of the program. Often cut is indispensable to achieve efficient execution.

A large number of Prolog builtin predicates thus provides extra control, Input/Output, database operations, arithmetic, term comparison, meta-logical operations, and set operations. Note that actual features vary between different Prolog systems. ISO supported the development of a standard for basic functionality in Prolog [45]. In practice, most implementations do not fully adhere to the standard, and all have extensions. Readers should refer to the specific Prolog manuals, such as SICStus Prolog's [7], ECLiPSe's [52], or YAP's [42] for the ultimate information on what is available on a specific system.

Note that the use of many built-ins, not just cut, relies on prior knowledge of Prolog execution. For example, a typical top-level clause for a program might look like this:

```
top_level :-  
    read(Query),  
    solve(Query, Solution),  
    write(Solution).
```

The correct execution of the built-ins `read/1` and `write/1` implicitly assumes left-to-right execution.

For more information about the relationship of Prolog and logic programming in general and for a thorough introduction to Prolog please consult [133].

2.2.1 Prolog History

Prolog's rules stem from activity in theorem proving in the early 1960's. This effort culminated in 1965 with the publication of the historical paper by Robinson [109] which introduced the now well known resolution rule, as an inference rule which is particularly well-suited to automation on a computer.

The programming language Prolog was born out of a project aimed at processing natural language. The name "Prolog" was invented in Marseilles in 1972. Phillipe Roussel chose the name as an abbreviation for "PROgramation en LOGic" to refer to the software tool they had designed to implement a man machine communication system through natural language. Arguably, Prolog was the successful result of a "marriage" between natural language processing and automated theorem-proving [34].

The theoretical basis for the then just invented Prolog language relies mostly on Kowalski's early work [82]. This pioneering effort gave a "procedural" interpretation to a set of Horn-clauses by showing that the application of the Robinson's resolution inference rule in proving a theorem in a horn-clause based system is equivalent to perform a computation.

In 1977, D.H.D Warren developed DEC-10 Prolog, the first Prolog compiler. This system brought Prolog performance to a level comparable to the best Lisp implementations of the day [150]. This helped to attract a wider following to Prolog and made the syntax used in this implementation and other implementations of Prolog developed at Edinburgh, such as C-Prolog [101] the *de facto* Prolog standard.

In 1983, Warren proposed a new abstract machine for the execution of compiled Prolog code which he called the "new Prolog engine" [151]. At about the same time, register based microprocessors became widely popular and led partly to the success of the abstract machine and to its great popularity among Prolog implementors. This abstract machine became known as the *WAM*, Warren's Abstract Machine, and it is until now by far the most widely popular way of implementing Prolog. Most development works, including those related to parallel Prolog, build on the WAM and/or extend it as the base mechanism for compilation and execution [147].

The Japanese Fifth Generation project [77] increased interest in both Prolog and parallel Prolog. The project took place at the eighties and closed at the early nineties without achieving its original goals. Prolog is nowadays mostly used for Artificial Intelligence style of applications, including Natural Language Processing, Machine Learning, and Expert Systems. In the nineties, the rise of constraint logic program-

ming opened new applications in areas such as Decision-Support Systems. Novel logic programming applications are reflected in conferences such as PADL (Practical Aspects of Declarative Programming) and the Practical Applications Series, include Model Checking, Software Engineering, and data-processing for the Web. Some of these applications are computationally very expensive, and thus would benefit from parallelism.

2.3 Sequential implementation of Prolog

The Prolog language adapts well to conventional computer architectures. The selection function and search rule are simple operations, and the fact that Prolog only uses terms means that the state of the computation can be coded quite efficiently.

2.3.1 The Beginnings

The original Marseille Prolog [34] system was an experimental interpreter, written in Algol-W by Philippe Roussel. A second interpreter was written in Fortran by Battani and colleagues. The system already provided built-ins that included the basic functionality available in modern Prolog systems.

Warren's DEC-10 Prolog system [148] was the first compiled Prolog system. The system was developed by Warren, F. Pereira and L. M. Pereira. The system showed good performance, comparable to the existing Lisp systems [149]. It included a separate stack to store terms. Mode declarations were used to simplify compilation.

The DEC-10 Prolog system became very popular. It is the reference for the “Edinburgh syntax” that is still followed by most Prolog systems. The efficiency of this system was also influential in the decision of the Japanese to use logic programming for their Fifth Generation Project.

2.3.2 The WAM

The basis for most of the current implementations of logic programming languages is the Warren Abstract Machine [151], or *WAM*, an “abstract machine” useful as a target for the compilation of Prolog because it can be implemented very efficiently in most conventional architectures. The WAM was developed out of the interest in

having a hardware implementation of Prolog. Warren presented a set of registers, stacks and instructions that could be efficiently implemented by specialised hardware. In practice, most WAM-based systems emulate such a machine in software, through an *emulator*.

The WAM represents Prolog terms as groups of *cells*, where a cell can be either a value, such as a constant, or a pointer. Variables are represented as a single cells. Free variables are represented as pointers that point to themselves. Bound variables can simply receive the value they are assigned to, if the value fits the cell size, or made to point to the term they are bound to. The WAM uses a *copy* representation for compound terms. In this representation a compound term is represented as a set of cells, where the first cell represents the main functor, and the other cells represent the arguments. Unification first compares the two main functors and then is called recursively for each argument.

The WAM was designed as a register based architecture. Arguments are passed through the A registers. These registers also double as temporary registers, known as X registers. Several other registers control the execution stacks:

- The *Environment Stack* tracks the flow of control in a program. An environment frame consists of the value cells for the permanent variables in a clause, together with a *continuation* which comprises of a code pointer and environment pointer that specifies the clause activation to return to on successful execution of this clause. An environment is created when a clause with more than one body goal is entered and is discarded before entering the last goals in a clause body (this is known as *last call optimization*). The E register points to the current active environment.
- The *Choice-Point Stack* stores open alternatives. A choice-point contains the necessary information to restore the state of the computation back to when a procedure was entered, plus a pointer to the code for the next clause to try, should the current clause fail. It is created (a **try operation** when a goal is entered which has more than one candidate clause, before the first clause is tried. On backtracking, the most recent choice point is taken and various registers are returned to their values as given by the choice-point. Variables trailed since the creation of the choice-point are reset and the next alternative clause is tried. If the clause being entered is the last clause to try for this goal then the choice-point is removed before entering the clause (a **trust operation**), otherwise the choice point's code pointer is updated to point to the clause following the one being

entered (a `retry` operation). The B register points to the active choice-point, which is always the last.

- The *Global Stack* or *Heap* was inherited from the DEC-10 Prolog abstract machine. It stores compound terms and variables that cannot be stored in the environment stack. The H register points to the top of this stack.
- The *Trail* stores *conditional* bindings, that is, pointers to variables whose values must be reset to “unbound” on backtracking. The TR register points to the top of this stack.
- The *Code area* contains instructions (and other data) comprising the compiled form of the program.
- The *PDL* is a push down list used for the general unification routine and is used as a call stack.

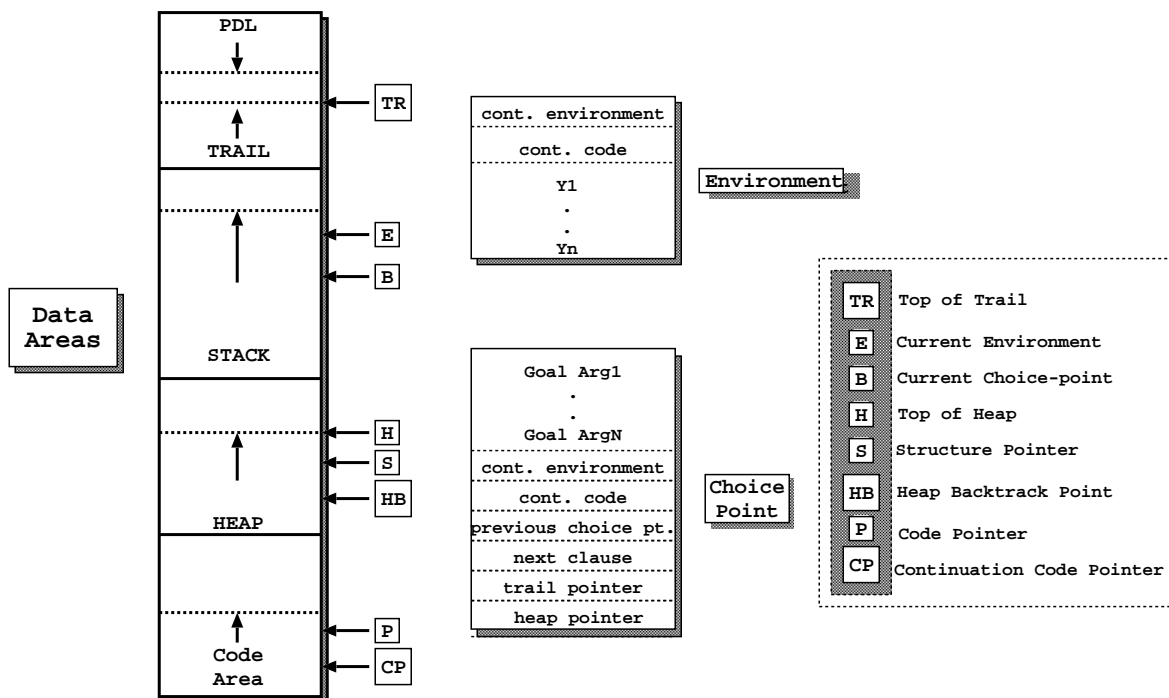


Figure 2.1: WAM data areas and registers

Figure 2.1 gives an overview of the stacks used by the WAM. The figure mentions other important WAM registers. The S register is used when unifying compound terms, and always points to the global stack.

Systems that implement the WAM, compile programs as a sequences of abstract machine instructions. To give the reader a flavour of what to expect from WAM code, we give a simple example for the naive reverse procedure:

```
nrev([], []).
nrev([H|T], R) :-
    nrev(T, R0),
    conc(R0, [H], R).
```

The WAM code for this procedure is shown in Table 2.1. The code shows examples

switch_on_term	CV1,Cc,C1,fail	
CV1:		
try_me_else	CV2	% nrev(
Cc:		
get_constant	[],A1	% [],
get_constant	[],A2	% [])
proceed		
CV2:		%
trust_me		% nrev(
allocate	3	
get_list	A1	% [
unify_variable	Y1	% H
unify_variable	X1	% T],
get_variable	Y2,A2	% R) :-
put_variable	Y3,A2	% nrev(T,R0)
call	nrev/2	% ,
put_unsafe_value	Y3,A1	% conc(R0,
put_list	A2	% [
unify_value	Y1	% H
unify_constant	[],	% [],
put_value	Y2	% R)
deallocate		
execute	conc/3	% .

Table 2.1: WAM code for Naive Reverse

of the four different groups of WAM instructions:

- *Indexing* instructions choose clauses from the first argument. An example is the first instruction in the code, `switch_on_term`. The instruction tests the type of the first argument and jumps to different code according whether the first argument is a variable, constant, compound term, or pair. Other indexing instructions switch on the value of constants and functors.
- *Choice-Point Manipulation* instructions manage choice-points. The code includes a `try_me_else` instruction, that creates a choice-point, and a `trust_me` instruction, that uses and then discards an existing choice-point. The `retry_me` instruction, not shown here, just uses a choice-point.
- *Unification* instructions implement specialised versions of the unification algorithm. The instructions are classified by position and type of argument. Head unification is performed by `get` instructions, sub-argument unification by `unify` instructions, and argument preparation for calls by `put` instructions. The `_variable` instructions process first occurrence of variables in the clause, `_value` instructions process non-first occurrences, the `_constant` process constants in the clause, the `_list` instructions process lists, and the `_structure` instructions process other compound terms.
- *Control* instructions manage forward execution. The `allocate` and `deallocate` respectively create and destroy environments. The `proceed` instruction returns from a fact. The `call` instruction calls a non-last subgoal, and the `execute` instruction calls the last subgoal. Note that by using the `deallocate` and `execute` instructions the WAM can perform last-call optimisation.

The outside simplicity of the WAM hides several intricate implementation issues. Complete books, such as Aït-Kaci's tutorial on the WAM [1] have been written on this subject.

2.4 Other Logic Programming Languages

One of the most serious criticisms of Prolog is that the selection function used by Prolog is too restrictive. From the beginning, authors such as Kowalski [83] remarked the effect on the size of the search space of solving different goals in different sequences.

A more flexible execution than the one used by Prolog can be obtained through coroutining. *Coroutines* cooperatively produce and consume data [83], allowing for

data-driven execution. Several designs for coroutining became very influential in the logic programming community. IC-Prolog, designed by Clark and others [31], was one of the first logic programming languages to support delaying of goals until data is available. Colmerauer's Prolog-II [32] supported `geler`, a built-in that would delay a goal until a variable would be instantiated. Naish's MU-Prolog and NU-Prolog [100] supports `wait`, a form of declaring that a goal should only execute if certain arguments were instantiated. Features similar to `geler` and `wait` are now common in modern logic programming systems.

Coroutining allows more flexible execution of goals. One can go one step further, and associate specific rules with variables. Whereas in traditional logic programming variables are associated with terms, in these novel frameworks variables may also be associated with real and rational numbers, intervals, booleans, lists, and so on. A special class of goals, or *constraints*, manipulates these variables.

The concept of constraint predates logic programming, but constraint logic programming has shown to be a very effective form of applying constraints. Initial work originates from Marseille group's Prolog-III [33], Jaffar and other's CLP framework of languages [79] and ensuing CLP(\mathcal{R}) system, and ECRC's CHIP [46]. There has been very intense research in the area since. We refer the reader to Marriot and Stuckey [93] for a good introduction to this rapidly expanding field.

2.5 Prolog as a parallel programming language.

Logic programs have non-determinacy in its selection and search rules, thus providing many opportunities for the exploitation of parallelism. Prolog inherits some of these opportunities.

The following types of parallelism have been identified and successfully exploited in logic programs:

Or-Parallelism: Results from the non-determinism present in the *select_{clause}* rule of SLD-resolution. When unifying a query goal with a head of a program clause, there can be several successful candidate clauses in the program. In sequential Prolog each candidate clause is tried in program textual order, through backtracking. With ORP they can be tried in parallel.

And-parallelism: Results from the non-determinism present in the *select_{literal}* rule

of SLD-resolution. In sequential Prolog the goals in a query are evaluated from the left to the right. However, in logic programming the nondeterminism present in the *select_{literal}* rule allows us to evaluate these goals concurrently and in parallel. Two main forms of And-parallelism arise :

- *Independent and-parallelism*: Occurs when the and-parallel execution of one goal does not interfere with the and-parallel execution of another. This can only happen if these goals do not share variables (strict independence) or in case they do, the bindings to the shared variables do not restrict the way those goals can be executed (non-strict independence) [76].
- *Dependent and-parallelism*: In this case goals are allowed to execute in parallel even when it cannot be known in advance that they will not affect each other if executed in parallel. This is the form of parallelism exploited in the successful family of concurrent logic languages [120]. This kind of parallelism is also known in some contexts as *stream* and-parallelism because dependent goals in a conjunction are executed in parallel with the value of the shared variables being *incrementally communicated* between them.

Unification Parallelism: In this case the arguments of terms being unified are unified in parallel. This is in general very fine grained and theoretical results have shown that the unification process contains some inherently sequential components [11]. This kind of parallelism will not be pursued in this thesis as it is so fine grained to require special hardware support.

It should be noted that “full” Prolog includes extra-logical features with side effects. These features are highly dependent on the sequential order of execution, thus removing opportunities for parallel execution. Special care must be taken with their implementation in a parallel Prolog system. Features with side-effects include input/output primitives, database primitives like assert/retract and the control feature “cut”. Often, proposals for parallelism are first based on a subset of Prolog which we shall refer to as *pure-Prolog*. Pure-Prolog contains no primitives which have side effects like those listed above. Other designers have instead specified new logic languages with special control facilities specially designed to guide parallel evaluation [30, 139, 120, 157].

2.6 Implicit Parallelism in Logic Programs

The main focus of our work is an And/Or parallel implementation for full Prolog. The parallel languages presented here are for reference and comparison with our approach in parallel logic programming.

Parallel logic languages vary from those which define a syntax, procedural semantics and provide special constructs for controlling the parallelism to those which assume Prolog as the logic language with no special facilities for the programmer to control parallel execution. The latter approach is termed implicit parallelism as the programmer does not have to be aware of parallel execution when writing his programs. The execution mechanism will automatically discover and explore the parallelism present in each application.

In explicit systems special types of goals (events and splits in Delta Prolog [102]) are available to control parallelism. Unfortunately, these languages do not preserve the declarative view of programs as Horn clauses, and thus lose one of the most important advantages of logic programming.

Implicit parallelism can be obtained through the parallel execution of several resolvents arising from the same query, *or-parallelism*, or through the parallel resolution of several goals, *and-parallelism*. These two forms of parallelism can be explored according to very different strategies. A large number of parallel models and systems have been developed for both distributed and shared memory parallel architectures.

2.7 Or-Parallelism

Or-parallelism is one of the most successful form of parallelism available in logic programming. It is amenable to efficient implementation and it has been shown to achieve very good speedups for a large range of applications, namely applications that require search, such as classical generate-and-test algorithmic solutions for complex problems, compilers and any problem whose solution involves search in a tree.

There are two fundamental problems that have to be tackled in any or-parallel Prolog system. These are (i) or-scheduling and (ii) variable binding representation. In this thesis we do not deal specifically with or-scheduling problems but instead draw upon the extensive work and literature available on the subject [70, 13, 23].

The problem in variable binding representations for or-parallelism arises because the same variable may have several different bindings in different or-branches of the computation tree. A number of different approaches [63] have been presented to tackle this problem. Arguably, the two most successful approaches for variable bindings representation are *environment copying* as used in Muse [3], and *binding arrays* as used in the Aurora [91] or-parallel Prolog systems.

In what follows computations are done by processing agents, that we will call *workers* or *agents*. More specifically we use and-agents in the context of an and-parallelism and or-agents in the context of an or-task.

We first describe the execution model followed by most ORP systems. We next survey the most successful variable binding representation schemes presented in the literature so far. Last, we briefly discuss scheduling issues.

2.7.1 The Multi-sequential Model

ORP systems execute tasks (or-work) in the search tree. Each task corresponds to an unexplored alternative of a choice point. Most shared-memory ORP systems are *multi-sequential* [92], that is, they try to minimise overheads by working as standard Prolog engines as much as possible. To do so, multi-sequential systems divide the search tree into a *public* and several *private* branches. Each private branch of the computation tree belongs to a single or-agent, while the public tree is shared by all. The key advantage is that exploiting alternatives in the private part of the search tree has much less overheads than executing in the shared part.

Multi-sequential systems are thus designed so that or-agents try to execute their private trees as long as possible, or, in other words, to maximise grain-size. Even so, or-agents eventually will run out of work. *Idle* or-agents will try to take work from the public part of the tree in order to start a new private branch within the or-agent stacks. The newly created private or-work will either be fully executed by its owner or will itself be made public. This process is either voluntary, say when the or-agent suspends work, or more frequently caused by requests from idle or-agents. Or-agents *publish* work by making one or more of its private choice-points visible to the other or-agents.

Last, note that the public part of the tree must itself be a tree. Idle or-agents will have to roam this sub-tree looking for new or-work. Private choice points are thus

made public in contiguous block, that always reach up to the border of the private and public regions.

2.7.2 Environment Copying

In the copying approach, each or-agent maintains its own copy of the current branch of the computation tree, e.g, each worker has a private copy of the heap, trail, environment and choice-point stacks. This solves the binding problem because each or-agent that can access a given variable has a private copy for the variable's cell. Every time a variable is bound, its value is thus saved in the private variable cell owned by the or-agent making the binding.

Sharing is performed by copying stacks from the worker that is giving the work to the worker that is moving down the tree. This copy is incremental as only the part of the branch that is not common between the two or-agents needs to be copied. After copying, some untrailing needs to be done, but this does not present any difficulty because the trail is also copied in the process.

This copy operation preserves memory references because the different stacks are copied to exactly the same memory positions in each one of the or-agents private stacks. This guarantees that no pointer reallocation needs to be performed when or-work is installed through copying.

The main advantage of copying is that it makes it possible for each or-parallel computations to be carried out in quite the same way as sequential execution is done in the WAM. Arguably copying is the most efficient way to maintain parallel environments in ORP. In the case of shared memory machines, the copy of large contiguous blocks of memory is an extremely efficient operation. As we shall see, this may no longer hold true in a combined and-or system because copying now has to deal with a large quantity of not so easy to identify smaller memory segments.

Copying was first used in the Japanese Kabu-Wake system [94] system (that was later abandoned in favour of the Fifth Generation Project) and in Ali's BC-machine [2]. Ali and Karlsson eventually adapted copying to standard shared memory machines, and developed the Muse system [3]. Copying was the basis for ACE [61], an And/OR parallel Prolog system.

In Muse, data sharing between or-agents happens through an auxiliary data structure associated with choice-points. This sharing point makes this approach amenable to

implementations in distributed memory computer architectures. For more details the interested reader can consult [81].

2.7.3 Binding arrays for or-parallelism

In the *binding array* (BA) approach all stacks are shared. In order to obtain efficient access to variable bindings each or-agent maintains a private data structure, the binding array, where it stores its conditional bindings for or-shared variables. To allow for quick access the binding array is implemented as an array, indexed by the number of variables that have been created in the current branch, from the root of the tree till the point where the variable is created. This index is stored in the variable's cell making access to variable bindings a constant-time operation.

The SRI model [152] was the first to propose using binding arrays for or-parallelism. In this model, execution stacks (heap, choice point, environment and trail) are shared by all system workers. During execution, a search tree is built with branches being dynamically distributed between the stacks of all the workers. This way, cactus stacks are formed for the system heap, environment, choice-point and trail stacks.

Each worker keeps a private set of bindings for shared variables. The key idea is that for each variable there is a single binding array slot identifier that is the same for all or-agents. This allows for fast access to a variable's local binding. Figure 2.2 presents the concept. In this case, variable X receives two different bindings in two different computation branches exploited respectively by worker W_1 and worker W_2 . The variable cell is shared by the two workers, so the bindings must be stored in a private location. In the case of the SRI model this location is each worker's BA.

The SRI model implements the BA by making the contents of each X variable cell to be the number of variables N between X and the root of the computation tree. In order to find out the private binding for the variable X pointed to by the reference $SHARED_PTR$, it is sufficient to search for the following value:

$$PRIVATE_VALUE = BA_BASE[* (SHARED_PTR)]$$

where $PRIVATE_VALUE$ is the binding for variable X . If this value is 0 then the variable is unbound. BA_BASE represents the first address or memory location of the binding array, and $SHARED_PTR$ represents the position of the shared variable, with $*(SHARED_PTR)$ giving X 's cell value.

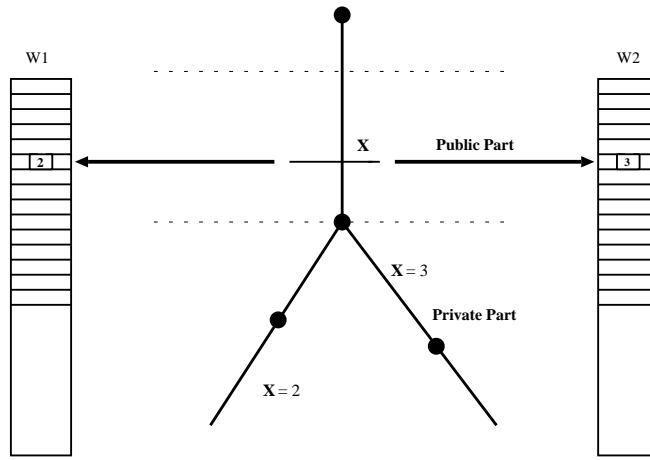


Figure 2.2: Binding Arrays in or-parallelism.

This operation is still a constant time operation, giving fast access to variable bindings. We will see later how $*(SHARED_PTR)$ can be seen as the variable's age. This makes it straightforward to detect when to trail and provides an easy way to guarantee that newer variables always bind to older variables.

2.7.4 Hash Tables

The main characteristic of hash-table models [28] is that whenever a worker conditionally binds a variable, the binding is stored in a *shared* data structure associated with the current or-branch (these data structures are implemented as hash-tables for speedy access). Whenever a worker needs to consult the value of a variable, instead of consulting the variable's cell immediately it will look-up the hash-tables first. Analysis of the PEPsSys model showed that a maximum of 7% of the execution time is being spent in dereferencing through the hash tables.

The PEPsSys system was developed at the ECRC research labs. The idea was to support both ORP and ANDP, but only a limited form of and-parallelism was eventually implemented. ECRC also maintained several Prolog and CLP systems. Although results were good, PEPsSys was never available outside ECRC, and ECRC's ECLiPse system uses copying [52]. More recently, Shen proposed hash tables in the style of PEPsSys in his FIRE system [125].

2.7.5 Or-scheduling

It is often neglected that parallel systems rely on a good distribution of the available parallel tasks. Dynamic task *scheduling* is known to be an NP-hard problem for all but the simplest cases. Such systems therefore rely on heuristics to obtain good performance. In the case of ORP, an ample body of literature describes several different approaches available and studied so far to tackle the problem [19, 23, 69, 14, 15, 131].

The scheduling problem is further complicated because or-parallel tasks may be in the scope of a **cut** further the left in the computation tree. If the cut executes, such or-tasks become “useless”, as they correspond to computations that would never have been executed in a sequential system. We therefore call this form of or-work *speculative*. A good scheduler must be able to avoid speculative tasks as much as possible whilst guaranteeing the maximum grain-size for each or-task. It has been argued [70, 15] that the best policy for or-scheduling in shared memory machines is to :

- Dispatch or-work on the *bottom-most*: by always publicizing the full extension of a private branch whenever an idle or-agent asks for work the idle agent has access to more or-work from the bottom part of the tree. Thus, or-agents will run out of work less often. Furthermore, because idle workers get work from the bottom of the tree they can quickly find extra public or-work, just by backtracking in the part of the tree that has just been made public. This is extremely important because traveling from branch to branch is much more expensive than just going up a branch.
- Concentrate on the left of the tree: by giving higher priority to or-work on the left part of the search tree, we will run into less speculative work. Furthermore, the scheduler should be capable of detecting that we are executing speculative work and that less speculative work is available elsewhere. In such cases, the scheduler should be able to suspend the current computation and move the or-worker to a less speculative or-task.

There is some discussion on whether such heuristics would hold for distributed machines, with several authors favoring *top-most* scheduling for distributed systems [130]. Their argument being that such a policy will in general increase grain size. Moreover, the size of the public part of the search tree will be less with this policy, and we will have more private work, further reducing overheads.

2.8 And-Parallelism

Whereas workers in or-parallel computations attempt to obtain different solutions to the same query, workers in and-parallel computations collaborate in obtaining a solution to a query. Each and-parallel task will contribute to the solution by binding variables to values.

The main difficulty in and-parallel arises when two parallel goals bind the same variable. This is called *variable sharing* and may occur if one or more and-parallel goals share a common variable in their execution. Consistent bindings for this variable must be enforced by the system if meaningful results are to be derived. On the other hand, all bindings for this variable must also be derived if the Prolog semantics are to be preserved in the parallel system. Complex machinery may have to be built in order to cope with these requirements. An alternative, is to avoid sharing of variables, as in “Restricted and-parallelism” [44].

Restricted and-parallelism simplifies the implementation of full Prolog semantics, including backtracking and side-effects, that would have been difficult to achieve if the whole picture of unrestricted and-parallelism had been tackled at once.

A second argument for Restricted And-Parallelism is that unrestricted and-parallelism may lead to too much synchronization among tasks making its parallel execution no longer viable.

A final argument for restricted and-parallel lies on the development of automatic Prolog code annotators. Experience has shown quite good results in detecting independence through static analysis.

The first design for restricted and-parallelism are the original *independent and-parallel* systems that only run goals that do not share variables in parallel. *non-strict independent and-parallelism* [76] gives a more general definition of independence between goals that allows for a less restricted form of variable sharing.

In contrast, *dependent and-parallel* systems allow goals that share variables to proceed in parallel, but (usually) enforce other restrictions. One example of such systems are the parallel implementations of the committed-choice languages. Parallelism in these languages can be exploited quite simply by allowing all goals that can commit to do so simultaneously. By their very nature, committed-choice systems do not have the multiple-solution problem, as they disallow multiple solutions. The binding conflict thus disappears, since if two goals in the current query give different values for the

same variable, the query is inconsistent and the entire computation should fail.

And-parallelism requires extensive changes to the system memory allocation scheme in order to support several workers operating concurrently in and-parallel. What makes these systems rather more complex to implement is that they also require support for the management of pools of available and-tasks, mechanisms for detecting the successful completion of and-parallel tasks, and the most complex of all, an efficient mechanism for the efficient propagation of failure [140], backtracking execution [75] and effective memory reuse.

2.9 Independent-And Parallelism

Independent-And Parallelism (IAP) represents a very useful form of parallelism that can be readily found in Prolog applications. IAP is mainly present in divide and conquer like algorithms.

DeGroot [44] proposed a scheme where only goals which do not have any run-time common variables are allowed to execute in parallel. To verify these conditions, DeGroot suggested the use of expressions that are added to the original clause and at run-time test the arguments of goals to verify independence. DeGroot's work on *Restricted And Parallelism* was later refined by Hermenegildo. Hermenegildo proposed the conditional graph expressions, or *CGEs* [71], and &-Prolog's parallel expressions [99] to control and-parallelism. We next give an example of a linear parallel expression in the &-Prolog language:

```
( ground(X), indep(Z, W) ->
    a(X,Z) & b(X, W) ;
  a(X,Z), b(X, W) )
```

If the first two conditions hold, the two goals `a(X,Y)` and `b(X,W)` are independent and can execute in parallel, otherwise they are to be evaluated sequentially. The `ground` condition guarantees that the shared variable will not contain unbound variables at run-time, and the test `indep/2` guarantees that `Z` and `W` do not share run-time variables.

Existing and-parallel Prolog systems (&-Prolog[74], &Ace[105] or DDAS[122]) use the same binding representation scheme as sequential Prolog. The reasons are that the Prolog scheme has proven itself in sequential Prolog implementations, and that it is compatible with and-parallel execution. Unfortunately, this principle does not hold if IAP and ORP are to be combined into a single system.

2.9.1 Extending the WAM for IAP

The RAP-WAM [72] is an extension of the WAM intended to support the forward execution semantics of CGEs. Upon arrival at a parallel call within a CGE, a scheduling mechanism will assign work to the available processors and novel data structures will keep track of the state of execution of parallel siblings.

A RAP-WAM based parallel system is composed of several RAP-WAM engines working in parallel and competing for the execution of goals made available in the “*Goal Stack*”. Figure 2.3 shows how the RAP-WAM implements IAP forward execution. The “*goal stack*” keeps these goal-frames. The state of CGEs is represented through *parcall frames* that are stored in the choice-point stack.

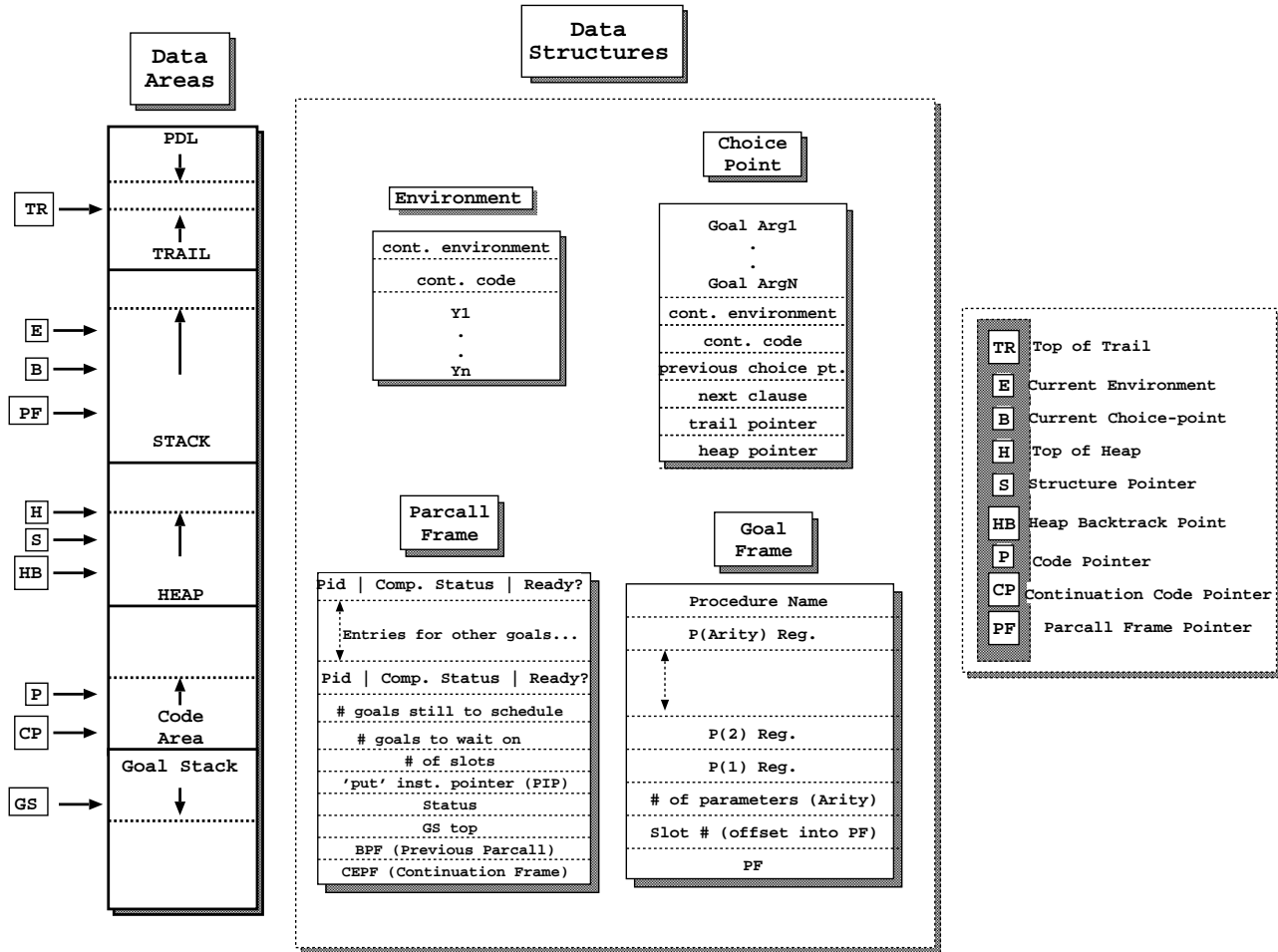


Figure 2.3: RAP-WAM data areas and registers

The goal stack We show a goal frame in figure 2.3. The frame contains all necessary information for the parallel execution of goals. In particular it contains the following items:

- **Procedure name** – Points to the first instruction of the procedure to be executed.
- **P(1)...P(n) registers** – Parameter Registers. A copy of the n argument registers for the goal call.
- **#parameters** – The goal's arity.
- **Slot #** – Identify the slot in the *parcall* frame that this goal belongs to.

Execution of a CGE starts with the tests. If they fail, the CGE is run sequentially. Otherwise, all goals in the CGE are pushed onto the *Goal Stack*. These goals can later be “*stolen*” by some other RAP-WAM engine which will copy the parameter registers into its argument registers, load its program counter P with the address of the “Procedure name” and start execution from there.

The parcall frame The *parcall frame* performs two major tasks:

- It controls forward execution of the parallel goals.
- It select the appropriate actions during backtracking.

A *parcall frame* is created for each parallel call. Moreover, for each parallel goal in the cge there is a *slot* in the parcall frame. A parcall frame thus has a variable number of computation slots, plus the following fields:

- **#goals still to schedule** – This field is initialized to the number of goals to be executed in parallel. Each time the local or remote processor takes a goal corresponding to this CGE from the *goal stack* this number is decremented.
- **#goals to wait on**– This cell is incremented each time a goal corresponding to this parcall frame is picked up from the *Goal Stack* for execution.
- **Total # of slots**–Indicates the total number of parallel goals in the CGE.

- **Put instruction Pointer (PIP)**– This is used during backtracking, to start pushing goals that were undone once again back into the *Goal Stack* for parallel execution. It contains the address of the first instruction of the first goal in the parallel call.
- **Status**– This cell marks whether execution of the parallel call corresponding to this *parcall frame* has already been completed once ((“outside”) status) or is still active with goals under it still yet to be completed ((“inside”) status). This is used to select the actions and type of backtracking that needs to be applied when this parcall frame is reached during backtracking.
- **GS**– The value of *Goal Stack* register upon entry to the parallel call. It is used in backtracking to restore the *Goal Stack* register (GS) before forward execution is resumed.
- **BPF**– A pointer to the previous *parcall frame*. It is used to restore the value of the PF register when the parcall frame ends up being backtracked over.
- **CEPF**– The continuation EPF. The value of the EPF register before this parcall frame is created is saved in this cell. It is used to reset EPF for the continuation of the CGE.

The RAP-WAM also introduced further features supporting goal-scheduling, memory management [73] and backtracking [75] that we survey next.

2.9.2 Backward Execution with CGEs

Backtracking in the presence of parallel goals can no longer proceed as in sequential Prolog. Hermenegildo and Nasr [75] identified two cases of backtracking in the presence of parcalls:

- **Inside backtracking**– Also known as *Conjunctive failure*, occurs when failure occurred before a solution was found for the CGE. Since all goals within the CGE are independent (they do not share any uninstantiated variables) we know the conjunction will surely fail. The only appropriate solution is therefore to terminate all other active goals within the same CGE and propagate failure to the most recent choice-point before the CGE. This way, backward execution in the presence of CGE’s incorporates a restricted form of intelligent backtracking.

This is directly supported in the RAP-WAM with the parcall **Status** field, which can be in **Inside** or **Outside** status.

- **Outside backtracking**– Alternatively, backtracking may be propagated to a previously terminated CGE. When this happens the following can happen:
 - If all goals within CGE had terminated without alternatives, they all must be undone and backtracking should continue towards the CGE’s parent choice-point.
 - If the CGE had with alternatives the rightmost nondeterministic goal must be found and this goal’s most recent choice-point B identified. All goals to the right of the rightmost nondeterministic goal must be restarted and forward execution resumed with an alternative from B. Before resuming forward execution the state of the parcall will be reset to **Inside**.

These two forms of backtracking extend the “most recent choice point” Prolog backtracking model to the independent and parallel execution model, preserving the generation of all “tuples” and offering parallel forward execution after backtracking.

2.9.3 &-Prolog

Hermenegildo’s &-Prolog system is an implementation of the RAP-WAM that supports Prolog. As in Prolog, the scheme recomputes the solutions to independent goals if previous independent goals are nondeterminate.

The &-Prolog language extends Prolog with the parallel conjunction and goal delaying. One of the system objectives was to have sequential execution as close to Prolog as possible. To do so, &-Prolog maintains much of the Prolog data structures. &-Prolog programs are executed by a number of PWAMs running in parallel [71]. The instruction set of a PWAM is the SICStus Prolog instruction set, plus instructions that include the CGE tests and instructions for parallel goal execution. Synchronisation between goals is implemented through the *parcall-frames*, data-structures that represent the CGEs and that are used to manage sibling and-goals. The resulting system shows good speedups for the selected benchmark programs, including examples of linear speedups.

An essential component of &-Prolog is the &-Prolog compiler. This compiler can use global analysis to generate CGEs. (Notice that for some programs this may still be

difficult, and &-Prolog allows hand-written annotations). Abstract interpretation [40] is used to verify conditions such as groundness or independence between variables are always satisfied. If they are, the CGE generator can much simplify the resulting CGEs, and avoid the overheads inherent in performing the CGE tests.

The &-Prolog system was designed to support full Prolog. Similar to Or-parallel systems, parallel executions of goals may break the Prolog sequence of side-effects. Several solutions have been proposed for this problem, the one actually used in &-Prolog is simply to sequence computation around side-effects.

More recently, Shen has re-implemented most of the functionality of &-Prolog in his DASWAM prototype for DDAS [124]. Pontelli and Gupta at New Mexico State University have also redesigned most of &-Prolog for their &-ACE prototype [106]. The &-ACE system has been very successful as an efficient implementation of independent and-parallelism, and has contributed with several important run-time optimisations [107].

2.10 The Committed-Choice Languages

Research into coroutining led some authors to completely abandon the Prolog's left-to-right execution. The committed-choice, or *concurrent*, logic programming languages [121] are a family of logic programming languages that use the process reading [146] of logic programs. In this reading, each goal is viewed as a process and computation as a whole as a network of concurrent processes, with interconnections specified by the shared logical variables. The process reading of programs is most useful to build *reactive* systems, which contrast with *transformational* systems in that their purpose is to interact with their environment in some way, and not necessarily to obtain an answer to a problem. Examples of reactive systems are operating systems and database management systems.

Initial research on these languages started in the early to mid eighties with Clark, Gregory and others' IC-Prolog [31] and then Parlog [30], and with Shapiro and others' Concurrent Prolog [119]. At the time, the Japanese Government was starting an ambitious research project on developing the Japanese hardware and software industry, the Fifth Generation Computing Systems Project (FGCS). Shapiro was influential in persuading Japanese researchers to use committed choice languages as a basis for this project. These languages were seen as an high-level programming tool that

naturally allowed the exploitation of concurrency and parallelism. Ueda's GHC [143], later simplified to KL1 [144], was the basis for FGCS work. The FGCS project had huge impact outside Japan, and both the American and the European governments supported alternative research on sequential and parallel committed choice languages and on traditional Prolog systems.

One major difference between Prolog and the committed choice languages is that clauses are *guarded*. The head and usually some goals in the body of a clause are tested before executing. If they are satisfied, one says that a goal can commit to the clause. If the goal does commit, *the remaining clauses are discarded*, even if they would match the goal. This simplified semantics and implementation. A further simplification resulted in the *flat* committed languages where one only allows a few built-in goals, such as tests or arithmetic conditions, in the guard.

Research in these languages was very intense during the eighties. Quite a few committed choice languages and dialects have been developed and rather sophisticated applications developed, especially within the FGCS [77] project.

The decision to support a single solution simplifies the design and implementation of these languages. Arguably, nondeterminism in choosing clauses is sufficient for most reactive systems, and indeed the committed-choice languages have been used successfully to implement complex applications such as operating systems kernels or compilers [120]. On the other hand, search programs that can be coded easily and naturally in Prolog are much more awkward to write in these languages [142]. Several authors proposed languages that allowed non-deterministic procedures in a committed choice environment, such as Saraswat's CP[↓,|,&,;] [118], Yang's P-Prolog [156], and Takeuchi's ANDOR-II [138].

2.11 Andorra Based Systems

Yang's work on P-Prolog [156] was an important influence in the David Warren's *Basic Andorra Model*. This model follows the *Andorra Principle*:

- Goals can execute in And-parallel, provided they are *determinate*;
- If no (selectable) goals are determinate, we can select one nondeterminate goal, and explore its alternatives, possibly in Or-parallel.

The model can be used to parallelise Prolog programs, or to design new languages. Work on Prolog parallelisation was mainly pursued at Bristol and resulted in the Andorra-I system [117]. Several other groups proposed novel languages based on this model, such as Haridi's Andorra Prolog [68], developed at SICS, and Bahgat's Pandora [10], from Imperial College.

Andorra-I was the first implementation of the Basic Andorra Model. The system was developed at the University of Bristol by Beaumont, Dutra, Santos Costa, Yang, and Warren [117, 155]. It was designed to take full advantage of the Basic Andorra Model. This means both exploiting parallelism, and exploiting implicit coroutining as much as possible.

Andorra-I programs are executed by *teams* of abstract processing agents called *workers*. Each worker usually corresponds to a physical processor. Each team, when active, is associated with a separate or-branch in the computation tree and is in one of two computation phases:

Determinate For a team, as long as determinate goals exist in the or-branch, all such goals are candidates for immediate evaluation, and thus can be picked up by a worker. This phase ends when no determinate goals are available, or when a determinate goal fails. In the first case, the team moves to the non-determinate phase. In the second case, the corresponding or-branch must be abandoned, and the team will backtrack in order to find a new or-branch to explore.

Nondeterminate If no determinate goals exist, the leftmost goal (or a particular goal specified by the user) is reduced. A choice-point is created to represent the fact that the current or-branch has now forked into several or-branches, while the team itself will explore one of the or-branches. If other teams are available, they can be used to explore the remaining or-branches.

During the determinate phase, the workers of each team behave similarly to those of a parallel committed-choice system; they work together to exploit and-parallelism. During the non-determinate phase, and on backtracking, only one particular worker in the team is active. This worker is called the *master* and the remaining workers *slaves*. The master performs choice-point creation and backtracking in the same way as an Or-parallel Prolog system.

Research on a more ambitious form of determinacy eventually led to the *Extended Andorra Model*, or EAM, where one can parallelise non-determinate computations

as long as they do not bind external variables. Warren was interested in the EAM as a way to exploit all forms of implicit parallelism in logic programs [153]. The EAM forms the basis for AKL [96] a successful And/Or-parallel system and for the BEAM [88] an implementation based on the original Warren's EAM proposal and the Parallel-BEAM [89], a parallel extension for the sequential BEAM.

2.12 Summary

Logic programming is an ideal paradigm for the development of parallel systems. Although the wide acceptance of parallel logic programming has been hampered by the late emergence of parallel computers, logic programming is one area in computing where mainstream systems do support implicit parallelism. In fact, or-parallelism is supported in the Prolog systems such as ECLiPse [52], SICStus Prolog [7], and YAP [42, 110]. More experimental, but still excellent, available systems include Andorra-I [117], KLIC [29], Penny [97], DASWAM [124], and ACE [106]. Open problems include how to efficiently combine and-parallelism with or-parallelism while preserving Prolog style of execution. In the next chapter we present previous research done in this area.

Chapter 3

Combined And/Or systems

In the previous chapter we described several executions models and associated implementations for ANDP and ORP. An important lesson that has been learned from the implementation and use of those systems and also from detailed simulation studies [126] is that Prolog programs can contain several forms of implicit parallelism. Systems that can exploit two or more forms of parallelism [158] at the same time should thus incur better speedups and obtain better absolute execution times. Ideally, they should perform as well as a non combined system when a single form of parallelism is present in a program and perform as efficiently as a state of the art sequential system when no exploitable form of parallelism exists in the program. Until very recently, few parallel Prolog execution models aspired at achieving this behavior.

In this chapter we describe attempts made by the parallel logic programming community to try to achieve combined systems. The development of combined Prolog parallel systems is made difficult by the apparent contradiction between ORP and ANDP regarding access to environments. In purely ANDP system an and-agent must *share* or at least be provided access to the other and-agents' environments, as all the and-agents are resolving the same conjunction of goals. In contrast, in a pure ORP system each or-agent must keep its environment values *private* to prevent binding conflicts.

Combined system thus have to support workers sharing the same environment (for ANDP) and workers not sharing the same environment (for ORP). A fairly satisfactory way of organizing these two different types of workers is to organize them into *teams* [117, 60]. Workers inside a team share the same environment and work in and-parallel. Different teams, each associated to a different environment, together form

the or-workers that compute the different branches of the search tree in or-parallel. In a nutshell, a team is an or-agent with a single environment that is shared by all its member and-agents.

A second problem that And/Or parallel systems must address is that alternatives created by and-agents must be made available for or-parallel execution. Unfortunately, this introduces problems to the environment representation schemes previously developed for pure ORP.

Regarding *copying*, the main problem now is what to copy. Copying works well because in systems that exploit pure ORP execution each branch proceeds in a contiguous stack. With ANDP execution we may need to copy goals that have been distributed through different data areas. The ACE system [59] proposed the first copying-based solution for this problem, based on the C-tree model [60]. ACE must tackle a few difficult problems. The major one is that at copy-time ACE must find out which data areas to copy, making copying more complex than, say, for Muse. It is also likely that more copying than for pure or-parallel systems will be required for efficiency reasons. Further, we shall see in section 4.4.2 that ACE is a not a *memory independent* based system. This prevents the immediate reuse of independent and-parallel computations between or-parallel branches.

The *binding array* approach simply does not work with combined systems. The problem is that the same binding array has to be shared between computations that execute in and-parallel. This creates a complex memory allocation problem. The first solution for this problem was the paged binding array PBA [64]. The PBA uses the operating system concept of *paging* to provide an elegant solution to the problem. Unfortunately, this solution incurs an extra cost that results from a more complex dereferencing and managing of bindings in the BA. We shall see in section 4.4.1 that PBA based systems are also not *memory independent*, which prevents the immediate reuse of all independent and-parallel computations between or-parallel branches.

An alternative scheme has recently been proposed by Shen [125]. The scheme uses hash tables, as Pepsys [28]. Hash-tables have a *non constant* time variable access scheme. Arguably [58], *constant time* variable access based models are more likely to result in more efficient systems.

We now proceed with a survey of proposed binding schemes for combined ANDP/ORP execution models. Later on, we will describe some of the associated problems we have found and that eventually lead us to develop our own solution to this problem. Throughout we focus on the simultaneous exploitation of IAP and ORP. We do not

consider other forms of and-parallelism as IAP and ORP are the ones closest to sequential Prolog. We proceed by summarizing in more detail the previously proposed binding schemes for combined ANDP/ORP systems.

3.1 Execution models for And/Or parallelism

Two major approaches have been proposed for combining independent and-parallelism with or-parallelism. *Reuse* and *recomputation*.

3.1.1 Reuse-Based Execution Models

In this approach the computation is seen as an And/Or tree. Solutions for each goal in a conjunction are stored and then combined with solutions from the other goals through a special operation, the *cross-product* calculation. This solution looks at first sight very attractive because it can reduce the search space of logic programs, and was the basis for initial proposals such as Conery's And/Or Process model [35] and Kalé's REDUCE OR process model [80]. A combined system for IAP/ORP where reuse is also proposed appears in Gupta's proposed And/Or model [58].

As it was argued and demonstrated in [62] reuse-based execution model have several difficulties. Namely, it becomes quite cumbersome and complex to maintain the sequential operational semantics of Prolog. This happens precisely because the systems lose the Prolog ordering of goals. Thus supporting side effects is quite difficult.

A second argument against reuse stems from a simulation study [126] that showed that typical Prolog programs perform very little recomputation. This is a result of the fact that Prolog programmers are aware of the recomputation characteristics of sequential Prolog, and thus write their programs in such a way as to minimize recomputation. This reinforces our view that the actual effort in implementing the cross-product operation would not, in practice, save a great amount of computation for real Prolog applications.

3.1.2 Recomputation-based execution models.

In this approach and-goals are recomputed in quite the same way as Prolog would. Major examples of such models are the the or-under-and and and-under-or [126] and

the C-tree [60].

In recomputation-based execution models, or-agents, or *teams*, work together in or-parallel to explore different branches of the search tree. Within each team, and-agents execute and-tasks as they would in a purely and-parallel based system, such as &-Prolog. Several different teams thus may each have an and-worker computing the same and-parallel goal in advance and they all generate and repeat the same solutions for this goal.

For an example consider the following code:

```
a :- b & c.
```

```
b :- d1.
```

```
b :- d2.
```

```
b :- d3.
```

If there are three teams in the system we can arrive at the situation illustrated in Figure 3.1, where goal *c* is being computed thrice in parallel.

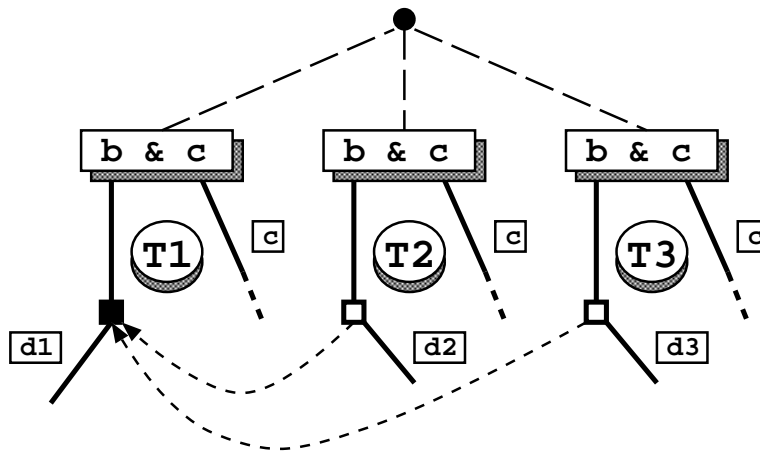


Figure 3.1: Goal *c* being recomputed thrice.

Observe that finding all solutions for *b* in the sequential execution of Prolog would also involve backtracking over *c* twice and thus recomputing *b* three times.

A recomputation based combined model thus ends up recomputing goals in much the same way as Prolog or &-Prolog would. It is not therefore surprising that recomputation-based models have some important advantages, namely:

- The operational semantics is analogous to Prolog's, thus simplifying the efficient support of side-effects in the presence of And/Or parallelism.

- The overheads associated with implementing cross-product nodes are avoided and saving the corresponding solutions for reuse is not necessary.
- Prolog programmers normally write their programs in such a way as to avoid recomputing solutions, resulting in little gains for reuse.
- Recomputation allows producer-consumer dependent and-parallelism [123, 104].
- A recomputation based approach overtly *simplifies* extending or-parallelism to incorporate and-parallelism.

3.2 Binding Arrays for And/Or Parallelism

And-workers in a team will share the same set of variable bindings. In BA based systems, this means that they will share the same binding array. The system exploits or-parallelism by having several of these teams active at the same time, each one computing a different branch, as in Andorra-I [117], or as in the C-tree model [60].

The question now is how to manage a binding array shared by several and-agents. The main problem is that several and-agents may want to place variable cells at the same binding array positions. Ideally, one should partition the binding array into correctly guessed sized chunks and allocate them to the different independent and-tasks. Unfortunately, it is impossible to know beforehand how many variables are going to be created in a computation. To do so we need to know whether the computation will terminate, which in the case logic programs reduces to the *Turing halting problem* [20], and is thus undecidable.

The problem is illustrated in Figure 3.2. On the left we see the situation for which the binding array was originally developed, sequential execution within an or-branch. In this case each binding array position (variable offset) has the following properties :

- The offset of a conditional variable is fixed for its entire lifetime.
- The offsets of two consecutive conditional variables in an or-branch are also consecutive

In this case the offset for the first variable of goal B is known because goal A completes before goal B is called. On the right we observe the case where goal A executes in

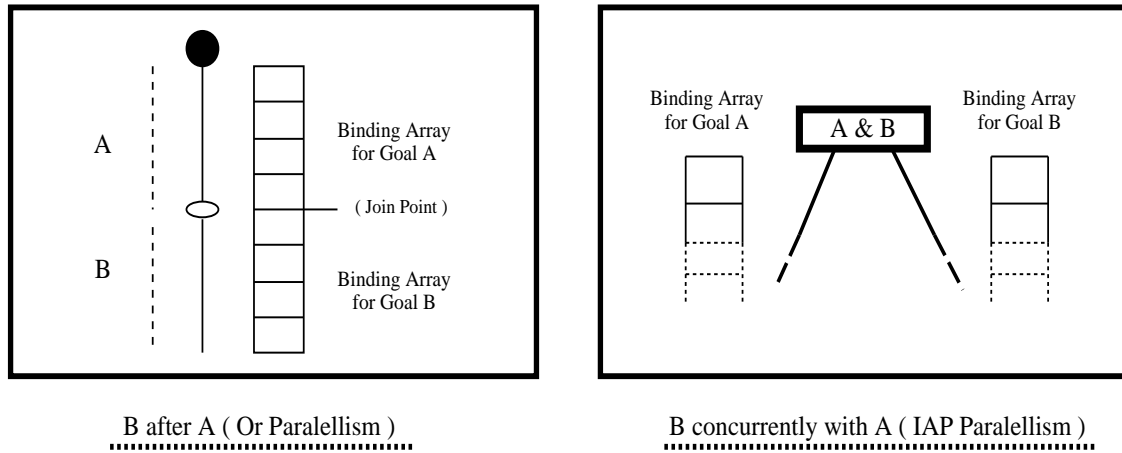


Figure 3.2: Using the binding array in and-parallelism.

parallel with goal B. Since we cannot know beforehand how many variables goal A creates there is no way we can estimate the offset for the first variable created while computing goal B [57].

A solution to this *offset* problem would simply be the maintenance of a team counter for variables indices. Whenever a new variable has to be created, we would lock the global counter, increment it, and the new value would be the new variable's offset. Unfortunately, the necessary synchronisation defeats the whole idea of having independent and-computations going on in parallel at maximum speed. Reusing BA entries after backtracking would also be very difficult.

Allocation of a common recurse between competing processes arises in operating systems, where physical memory has to be shared between different operating system processes. The usual solution to this problem is to paginate memory. The system first divides memory into frames, all with the same size. Each process then maintains a private memory page table that translates from the process's addresses to the physical memory addresses. In this way several processes can share same physical memory whilst maintaining their own private non clashing virtual memory addresses.

The paged binding array model (PBA) [57] was inspired by this idea to implement a similar strategy for binding array management in a team.

3.2.1 The Paged Binding Array

In the PBA model each shared variable cell contains two fields, the page table offset, *PG_NUMBER*, and the offset within that page, *OFFSET*, where the cell's variable is held. The position in the BA for the variable cell is determined by obtaining the page address from the page table, *PT* (indexed by the page table offset), and adding to this address the variable offset within that page. This process is illustrated in Figure 3.3. Each time a new page is needed, its BA slot is allocated from a list of free BA offsets. Observe that allocating a new page slot is equivalent to allocating a new page as there is a one-to-one correspondence between page offsets and BA pages.

The PBA thus uses indirection to tackle the binding array management problem with the page size being implementation defined. A page table manages page offsets with access synchronized among the elements of the team.

A new page is allocated whenever and an-agent:

- Creates a variable and there is no more space in the current page.
- It is starting the execution of a new task, be it, an or-task or an and-task.
- It has completed a parcall and is now going to execute its continuation.

The allocation of new pages involves three operations: **(i)** a new page of size K is allocated from the page table's free page list; **(ii)** a new page entry J is reserved from the page table; and **(iii)** a special trail entry is pushed in the trail stack, indicating the allocation of the new page slot. The first two operations, **(i)** and **(ii)**, must be atomic within the team.

The value held in a variable's cell (the *valuecell*) is the pair $\langle P, I \rangle$ where P is the index in the page table where the starting address for the BA page is kept and I is the offset within that page where the team value for the variable is held. The binding of the variable is then obtained as follows:

$$BINDING_VALUE = BA_BASE[PAGE_TABLE[P] + I]$$

Figure 3.3 illustrates PBA management. As we said, whenever a new page is allocated a special entry, *new_page*, containing the page offset, is "trailed". The new entry allows for page table entries to be reclaimed during backtracking and allows for other teams

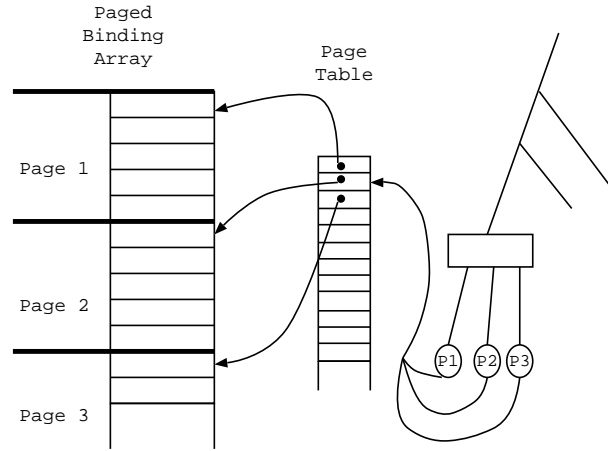


Figure 3.3: The Paged Binding Array.

to use the correct page table entries when installing or-work from this branch. We thus allow valuecells $< P, I >$ to be shared by several teams. Sharing starts as soon as the part of the branch that contains this cell is made or-public.

The key point of the PBA is thus to guarantee the value P corresponds to meaningful page table entries in all the teams that can access the same cell. To obtain performance we would like to allocate the minimal number of pages that would allow for sharing, whilst guaranteeing that different teams will not unnecessarily share pages. In practice, in order to guarantee correct execution we must:

1. Create at least one page per each goal in a parallel conjunction.
2. In order to preserve variable age we should allocate a new page before executing the continuation of a parcall frame, so that the offset for the new page will be higher then all of the page offsets used to execute the parallel conjunction we are closing in the parcall.

Note that as a result of item 2, the page table array can develop *holes*. The problem appears in the cases where a team has several and-agents competing for page offsets under a parallel conjunction and one of workers in the team, say W_b backtracks. The page offsets freed for W_b may not be reusable by other workers, if they are already using larger offsets. Moreover, when the parcall closes, unless ancestors to the current parallel conjunction are available, all pages with smaller offsets that have been released after backtracking, and that could not be reused are trapped: they can no longer be used in the continuation or in any other and-goal in this team. Only by backtracking over this parallel conjunction will they ever be freed.

One solution would be not to use the cell value for variable age. Alternatively, we could access holes as the price to pay for cheap calculation of a variable's age. A full assessment of the best solution would have to wait for an actual implementation.

A second difficulty with the PBA is choosing ideal page size: should we go for large page tables and small pages, or for large pages so that we have a smaller page table? What about variable page size, considering that many computations will have different granularities.

Our work with the PBA showed that the actual implementation of the PBA would be quite complex, and might incur in serious overheads.

3.3 Copying for And/Or Parallelism.

The copying approach for or-parallelism has been introduced in section 2.7.2 . In what follows we will summarily describe ACE [59] (And/Or-parallel Copying-based Execution), a copying based And/Or parallel implementation scheme for the execution of Prolog programs.

ACE is a C-tree based execution model that uses a variable environment representation scheme based on stack copying. ACE intends to subsume both MUSE and &-Prolog in terms of execution behavior. As in any C-tree derived model, all available workers are divided into teams in such a way that and-parallel work can only be shared between workers of the same team and or-parallel work is to be shared between teams. In ACE all teams share the same logical address space, independent for each team but with an identical address mapping in all teams to allow for copying without pointer relocation.

To implement and-parallelism, the address space of each team is divided up into k memory segments, where k is the maximum number of and-agents allowed per team. Each and-agent in a team allocates its stack set from its corresponding memory segment. Once the address space in a team has been divided in these k stack sets (that need not be identical in size) the same division must be done for all other teams of the system in order to prevent pointer relocation. And-agents belonging to one team are allowed to join a different team as long as the new team has a free memory segment where they can allocate their stacks. New teams can be created dynamically, but their workspace must mirror the other teams' and thus be divided in k segments. As in MUSE, a team installs or-work by copying stack segments. As we shall discuss next, in the presence of and-parallelism determining what to copy becomes a problem.

Copying is straightforward in ORP systems. Indeed, it is a very fast operation in Muse as:

- Each worker only copies a small number of large contiguous blocks of memory, one for each of the WAM stacks, and copying large segments of memory is a well supported operation in most hardware.
- Or-work is dispatched on the bottommost, so that copying is only done a few times.

ACE needs to cope with ANDP:

I— When exploring and-parallelism a team intermingles the execution of the and-goals in such a way that instead of having a contiguous stack, a team ends up with what has been called “spaghetti” stacks [127]. These are the stacks that need to be copied, and the problem lies in how to untangle the stacks in order to identify which portions of the and-agents stack sets need to be copied in order to properly install or-work.

II— In order to properly install or-work, the installing team must unwind bindings after copying the appropriate and-goals segments. The work to unwind is the one resulting from all independent and-computations that occur to the right of the or-work node starting from the original copying team’s position down to the source of or-work that is being installed. To achieve this, the installing team must obtain a copy of all relevant trail segments for all these and-goals at the moment of copy. This would be similar to copying stack segments, except that some of the trail segments are being expanded.

One simple solution is to *freeze* all members of a team when sharing. Sharing thus will involve all members of a team. A benefit is allowing several and-agents to participate in the copying operation in parallel. Unfortunately, freezing a team can be a costly and lengthy operation.

Suppose that in an ORP system, or-agent W_1 is at node a and wants to install or-work in node b from or-agent W_2 . In the copying approach for Or-parallelism, W_1 can move from node a and install or-work at node b in three different ways [4]:

- Total copy: copy the entire content of the stacks of the “giving” worker and backtrack to node b .

- Incremental copy: copy the contents of the stacks of the “giving” worker, but only below node a . Then backtrack to node b .
- Optimized incremental copy: copy only the stack segments between choice points b and a and the trail segment below b . Then unwind the bindings in that trail segment.

In Muse the difference between “Incremental Copy” and “Optimized incremental copy” is usually not very significant because Muse dispatches work on the bottom-most. In the presence of IAP, and even if the system dispatches on the bottom-most, there may be a huge difference between these two approaches to copying. Incremental Copying should be easier to implement, but too much can be copied. Optimized incremental copying only copies what is needed but it is very hard to determine exactly what needs to be copied. And in both cases the untrailing problem is by no means trivial.

Pontelli et. al [59] propose balancing excessive unnecessary copying in incremental copying with the excessive synchronization time in optimized incremental copying, by detecting at copying time which of the two options may give the best results. Their heuristic is: If the choicepoint from which the alternative is being taken is outside the scope of any CGE, or if it only is in the scope of completed CGEs, then use incremental copying. Otherwise, they suggest using optimized incremental copying.

Two of the main reasons for the efficiency of Muse are thus lost in ACE. It is therefore an open question whether it is possible to implement ACE in an efficient way. &-ACE [105] achieved arguably the most efficient implementation of IAP for Prolog so far, but unfortunately the ACE project has still not been able to prove that this model is capable of delivering the promised integration of ANDP/ORP that was at the origin of this proposal.

The difficulties with copying and untrailing were one of the reasons that we concentrated on the SBA for our work.

3.3.1 Copy on write.

Recent research in the combination of IAP and ORP have led to a new binding representation approach: the α COWL (copy-on-write design) [111]. In BA systems, the stacks form a cactus-tree representing the search-tree. Processors work by expanding the tips of this tree. Workers always use a shared pool of memory except when storing

private bindings. These are stored in a local data-structure, the *Binding Array*. In contrast the α COWL scheme uses a copy-on-write mechanism to do lazy copying. This approach elegantly supports both IAP and ORP.

The *Copy-On-Write* model, or α COWL, was proposed by Santos Costa [111] towards supporting And/Or parallelism. In the α COWL, in a fashion similar to environment copying, each worker maintains a separate environment. The difference is that whenever a worker wants to share work from a different worker, it *logically* copies all execution stacks. The insight here is that although stacks will be logically copied, they will be *physically* copied only on demand. To do so, the α COWL uses the Copy-On-Write mechanism provided by most modern operating systems.

The α COWL has two major advantages. First, it can copy anything. It can copy standard Prolog stacks, the store of a constraint solver, or a set of stacks for ANDP. Second, because copying is done on demand, the system does not suffer from the overheads of copying large, non-contiguous, stacks. This is an important advantage for ANDP computations. The main drawback of the α COWL is that the actual setting up of the COW mechanism can be itself quite expensive, and in fact, more expensive than just copying the stacks.

To actually support sharing of work in the α COWL the idea is that whenever an or-worker P accepts a work request from another worker Q, worker P forks a child process that will assume the identity of worker Q, whilst the older process executing Q exits. At this point, the new process Q has the same state as that of P. The process Q then is forced to backtrack to the same choice-point. Note that scheduling is realised in exactly the same way as for any other copying based model, that is through the use of a public tree of or-frames in shared space.

One argument against the α COWL is that `fork()` is a rather expensive operation. For programs which have parallelism of high granularity, one expects that the workers will be busy most of the time and the number of sharing operations be small. In this case the model is expected to be efficient. On the other hand, worse results should be expected for fine-grained applications. One alternative would be to use the `mmap()` primitive as an alternative to `fork()` to implementing copy-on write.

Note that the α COWL was influenced by our work in the design of the SBA. Indeed, the motivation of the α COWL was to use the design principles that we describe in Chapter 4 in the context of copying. Chapter 5 presents experimental performance results for the α COWL and compares the α COWL with other environment representations schemes.

3.4 The Fire Model

All IAP/ORP recomputation based models we presented so far have in common the fact that ORP bindings for variables in a branch of the search tree are private values associated with the or-agent, or team, that is computing the branch. They are either located in private Binding Arrays in the case of SRI derived models (Andorra-I, PBA, and as we present later, SBA), or in private stacks as it is the case of Muse derived models (ACE) or of lazy copying schemes like α COWL.

The Fire model proposed by Shen [125] makes a radical departure from these environment representations by placing the ORP bindings for variables in the execution tree itself, and thus accessible to all computing agents. Thus, there is no need to associate and-agents into teams in order to have access to the or-parallel bindings of a branch, as was the case with the previous combination models.

The Fire model extends the PEPsSys [28] or-parallel binding model for ORP. Shen [125] argues that because PEPsSys appears to be the most efficient scheme that stores alternative bindings directly on the search tree, PEPsSys is a very sound base to extend for ANDP. PEPsSys had also suggested supporting IAP via the reuse of and-goals [154]. But, because Fire relies on recomputation, it fundamentally differs from the PEPsSys model.

The main advantage of the Fire model is that task-switching is constant-time. The operation thus precludes the use of sophisticated schedulers as is the case for the other combination models. Furthermore, without teams a worker is no longer bound to get work only from within its team and is instead free to fetch work from anywhere in the computation tree. We thus avoid an extra layer of scheduling to control the migration of workers from one team to another [48], as required for the C-team based execution schemes. Indeed, migration of workers between teams is an expensive operation: this further restricts the grain size of tasks an agent is able to profit from, even if the agent is able to migrate from team to team looking for work.

In the C-tree, a team looks for or-work by traveling within the public or-nodes of the computation tree. This means that when a team moves from one node to the other, the two nodes must be connected by a fully connected path. As a consequence only complete work is shared with other teams. By associating the alternative bindings directly with the search tree this restriction does not appear within the Fire model: all workers are free to pick-up work anywhere within the tree.

Unfortunately, all this flexibility comes with a price. Hash-table based models do not have constant time access to variables. Furthermore, with the inclusion of ANDP, the continuation of a parcall may include a very large number of structures, in the order of thousands, that should be traversed in order to access the binding of a variable. Arguably, this situation is very unlikely if the number of computation agents is reasonably small, as it would be the case in a current real shared memory parallel machine.

The main problem however rests in the way the scheme manages or-under-and parallelism. Special ORP markers are needed to manage or-work that are all linked together in their left to right order. The or-markers scheme proposed to maintain nested CGE's is extremely complex and probably too expensive to maintain in order to implement or-under-and parallelism efficiently.

3.5 DAOS – Scalable And/Or Parallelism

The differences between shared and distributed memory machines have become less significant in the last few years due to distributed shared memory systems (DSMs) [84]. Work on parallel logic programming systems for hardware DSMs has given interesting results. Dorpp ORP system [130] achieved good performance on a DSM machine. More recently, Santos Costa et al [113] have analyzed Andorra-I system on a DASH-like simulated machine. Their numbers confirm that most read cache misses result from scheduling and from accessing the code area. Misses to the execution stacks (most part eviction or cold misses) varied from 8% on an ORP parallel application to 30% on an ANDP application. Only the ORP application has significant sharing misses for an execution stack, the choice-point stack, (60%) and these misses are associated with the fact that this stack is also used for scheduling.

DAOS (Distributing And/Or in Scalable machines) [27] builds on the previous analysis in order to present a new approach to an And/Or distributed parallel system that is based on the following observations:

- Most large data structures in logic programming systems are built in the execution stacks. We should take the best advantage of caches to reduce network traffic.
- Previous studies show high rates of sharing misses in scheduler data-structures. This suggests using explicit messages for scheduling.

DAOS is the first parallel logic programming model for distributed systems to embody these ideas. Its main contributions are:

- A binding data representation that should prove to be simple to implement and that adapts naturally to DSM techniques. This would allow the use of previously designed synchronization and scheduling algorithms.
- The combined use of DSM and message passing techniques in the same framework: DAOS innovates over shared memory parallel logic programming systems by explicitly addressing the distribution problems inherent to distributed scalable machines.

DAOS uses the Sparse Binding Array (SBA) that we present in Chapter 4 in order to manage bindings to shared variables. Each worker has a private SBA that is synchronized with the other workers SBA's through the trail, both when sharing ORP and IAP work. The trail is used both to propagate conditional bindings during exportation of and- and or-work, and to return conditional bindings performed during remote execution of and-goals:

- Whenever a team imports or-work, workers have to move up or down in the search tree. This is performed by copying the trail from the exporter and installing the bindings in their SBAs.
- When a worker receives an and-goal to execute, it starts its execution in its own memory address space. Any conditional bindings, or any bindings to variables created prior to the and-parallel conjunction, is stored in the SBA, and trailed. At the end of execution of the goal, the worker returns its trail to the exporter.

Note that in order to support And/Or parallelism the trail needs to be segmented.

As illustrated in figure 3.4, each DAOS *worker* consists of three modules:

- The *Engine* – responsible for the execution of Prolog code.
- The *Work-dispatcher* – controls the exportation of both and- and or-work. This module and the Engine have exclusive access to the Control Stack, Trail, and Goal stack.
- The *Memory-Manager* – controls the major execution stacks, namely the Environment stack and the Heap through a page-based software DSM.

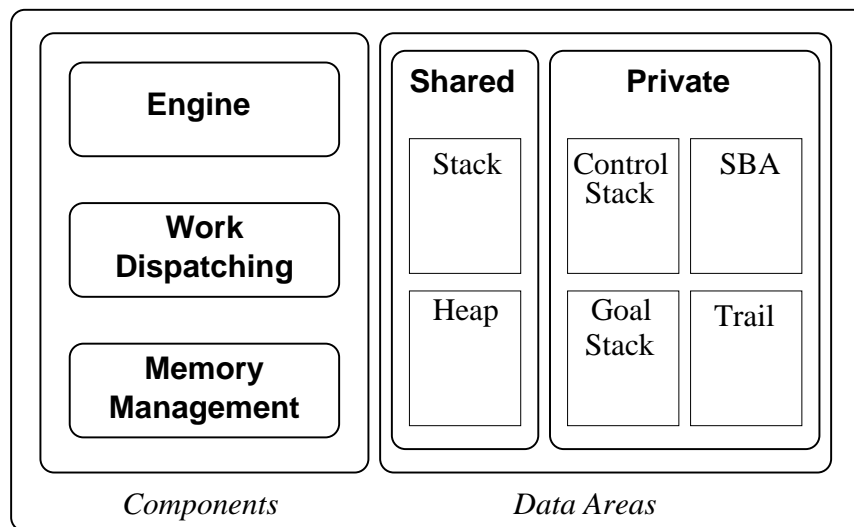


Figure 3.4: Daos Worker Organization.

The Engine module implements an SBA based abstract machine. It combines the use of SBA [37] to deal with or-parallelism, with mechanisms based on Hermenegildo's RAP-WAM [72] to deal with and-parallelism.

DAOS uses MPI to implement message passing and the commercial software TreadMarks [22] for DSM.

3.6 And/Or speculative work

Both ANDP or ORP may be *speculative*, that is, may correspond to parts of the execution tree that would not have been taken in a sequential execution of Prolog. More precisely, in ORP, an or-task is speculative [70] if, for instance, it is in the scope of a cut in the search tree. In ANDP, any goal to the right of a not yet completed goal that can fail, is speculative.

The degree of speculativeness of a task, be it an or-task or an and-task, is a very good indication of the usefulness of the computation being performed. It thus represents a valuable source of information to feed the schedulers (both or- and and-) in order to avoid the execution of unnecessary work that should not have been done in parallel in the first place.

To our knowledge, the special case of *speculativeness* that occurs in an And/Or system has not yet been aptly studied so far. To understand the problem we must study

the amount of extra computation done in the system as a whole when or-under-and parallelism is exploited and which does not occur when only one form of parallelism is exploited. To illustrate the problem we present the following example :

```
p :- goal(1000)

goal(N) :- N == 0, or_node.
goal(N) :- N > 0, N1 is N - 1, goal(N1) & large_computation.

or_node :- computation1, !.
or_node :- computation2, !.
or_node :- computation3.
```

If only ORP is exploited, the “&” is equivalent to “,” and the goal “large_computation” is run a thousand times in the same way as in sequential Prolog. If only ANDP is executed then the “large_computation” will be run a thousand times. However if or-under-and parallelism is allowed, then in a recomputation based system each alternative for the “or_node” predicate could have been grabbed by a different team. If no control at all is exerted in the way the relaunch of and-goals to the right is done when or-work is installed¹, we could end up running “large_computation” three thousand times as is illustrated in figure 3.5, when in all the other forms of execution, it is executed just a thousand times.

This is a direct result of the or-under-and model of execution. As in the other simpler cases, we think that this kind of speculative work cannot be totally eliminated without restricting the amount of exploitable parallelism too much. But it can, as in this case, be properly managed with an appropriate and-scheduling policy.

This form of speculative work can be present in programs that contain many or-branches inside CGE's of which only one lead to success. Simulation studies presented by Shen [126, 128] have presented such results. In [125], Shen argues against the use of teams by stating that they make it very cumbersome to control the execution of these highly speculative goals. We argue that with appropriate scheduling policies it is possible to greatly reduce this problem and at the same time maintain the concept of teams. We will later see that the concept of team will prove to be essential to our goal of maintaining IAP and ORP Prolog system as *orthogonal* as possible.

¹At the or-node corresponding to the clause “or_node”

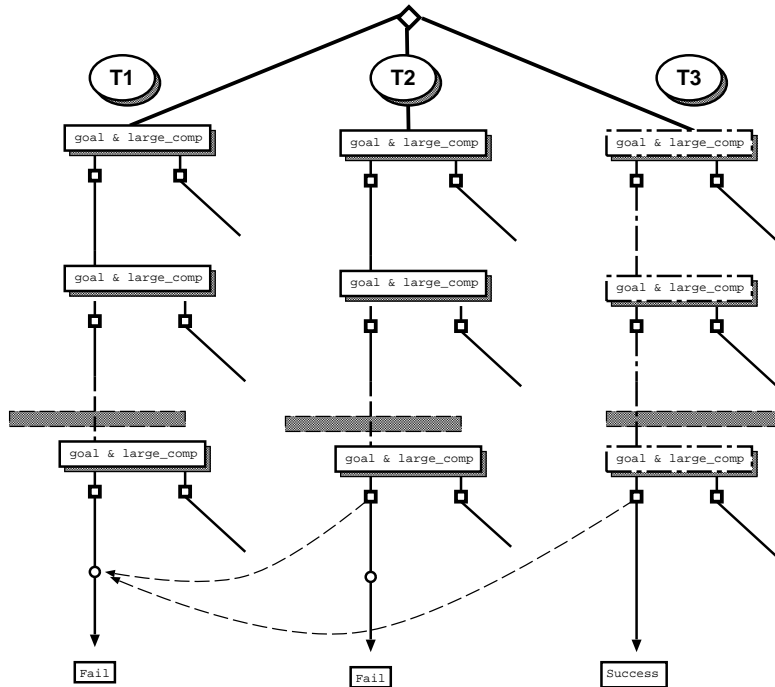


Figure 3.5: or-and speculative work.

We suggest two principles for And schedulers that want to address the problem:

- *Depth-first dispatching*: Give execution priority to goals deeper in the tree. If we have several goals from the same parcall, give priority to the leftmost goal. This way and-parallel execution will have a tendency to occur first to the less speculative left and deep down the And/Or execution tree.
- *Lazy goal dispatching*: only make parallel goals available for parallel execution when there is need to do so.

These ideas were incorporated in our LGSATS model [36], discussed in Chapter 7.

3.7 Summary

We have summarily described some of the original proposals for combined And/Or systems for the execution of Prolog. The presentation lays the foundations for our design of the SBA model, that we argue addresses some of shortcomings of previous

proposals. The interested reader should consult the extensive referred bibliography for further detailed information on each model.

In the next chapter we show how the *orthogonality* principle and the concept of *memory independence* are the keys to achieve efficient implementations of or-under-and parallel schemes for the execution of Prolog programs. We will also derive the sparse binding array (SBA), a data structure we present in the next chapter, that allows us to implement or-under-and parallelism in a very elegant and simple way.

Chapter 4

SBA and the Orthogonality Principle

In this chapter we present the SBA, a novel data structure designed to support And/Or parallelism efficiently. Our work in the SBA was motivated by a detailed study of the issues that make models such as the PBA [64] and ACE [59] hard to implement. We thus start by reviewing these issues and then proceed to present the main principles that, we argue, all And/Or parallel system should follow. We then apply these principles to design the SBA.

4.1 Orthogonality.

We next study how well the PBA and ACE fair in the simultaneous exploitation of IAP/ORP. First, we notice that the whole motivation of the C-tree was to separate the implementation of IAP and ORP. More formally, these models are first attempts at obtaining the

Definition 1

Principle of orthogonality – ORP execution should be unaware of IAP execution, and IAP execution should be able to ignore, as much as possible, the very existence of ORP in the system.

The motivation for orthogonality is that both ORP and IAP are hard to exploit. If we can separate the two forms of parallelism, it should be easier to implement

the combined system. Moreover, the less complex the implementation, the easier to optimise and thus, we believe, the lower overheads we will get.

The principle of orthogonal design is by no means novel. It is indeed normal practice in scientific and engineering endeavors to take advantage of techniques that have been used and proven effective in the past. Our goal is to apply this general principle to the design and development of a new combined Prolog parallel system.

4.1.1 Teams and Epochs.

At first sight, achieving orthogonality in And/Or systems would seem to be a difficult task. Memory access requirements for ORP seem to be the antithesis of what is needed in ANDP. Whereas in ANDP each worker must have access to every binding, in ORP system each worker should not have access to the other workers' environments. This follows because ANDP is associated with a specific branch of the search tree, whereas ORP is about exploiting different branches.

A solution to this dilemma is the concept of “team”, that we discussed before in Chapter 3. By organizing and-work and or-work hierarchically, teams cleanly separate the two forms of parallelism. We therefore envisage an IAP/ORP system such that ANDP is *encapsulated* within a team. We can thus imagine a team as a black-box that can exploit ANDP; and ORP as having several such boxes working together. Thus, after abstracting away the inner workings of a team, the team just computes its branch and “travels” along the computation tree in quite the same way as a purely ORP worker would.

The question is what happens within a team when it travels about the computation tree looking for work. Clearly one and-worker is active as the team is actually moving along the tree and the or-scheduler must be executing. The question is, what are the other elements of the team doing when a team travels? The following three possibilities arise:

1. Other workers may be active, but exploiting a different conjunction in the same or-branch of the computation tree.
2. Other workers will be idle, waiting for further and-work.
3. Idle workers can be *migrating* to some other team, in search of available and-work.

We say that a combined And/Or system is:

Definition 2

epoch-based — if we only allow a single team member to be active whenever the team is traveling along the computation tree looking for work.

The active worker is the and-agent executing or-scheduler code. All other and-agents either will be idle or will be migrating to other team(s). We say that each work sharing operation starts a new *epoch*: epochs start at a sharing operation and complete when a worker backtracks into a public node of the tree.

At first sight it seems that epochs still allow for orthogonality. If we are backtracking, shouldn't we kill everything within a team? In fact, this is not true, as epochs allow ORP to affect ANDP execution. As a result epochs may introduce serious waste and unnecessary complexity in a combined system.

The problem is quite intrinsic to the nature of IAP systems. ANDP work is work performed in advance, and recomputation-based IAP systems combine every solution for each goal in a CGE. That is, the first solution of the leftmost goal is combined with the goal to its right, and so on, until we combine *every* solution. Now, imagine that the public region of the search tree reaches the leftmost goal. Whatever solution we search in this parallel goal, computed solutions for goals to the right will be useful. Therefore, from the view-point of ANDP, there is no reason to kill work to the right if we are backtracking on a goal to the left, *even if this work happens to be or-public!* In fact, doing so breaks the very essence of IAP, as work that should be independent is now interfering. Epochs force all IAP work to stop, thus allowing ORP work to unnecessarily restrict IAP, and thus seriously break orthogonality.

The problem is further illustrated in figure 4.1. Team T_2 contains three and-agents, and-agent $W_{2,1}$ has backtracked and is installing or-work from choice-point c_2 . In a pure recomputation based system, and-computations under goal b in team T_2 would have to be rewound before this new or-work can be installed. This means that and-agents $w_{2,2}$ and $w_{2,3}$ must be stopped and rewind computations for goals d and e . After rewinding the computation under parcall (c, d, e) we will also have to rewind goal b . The process proceeds up to the root of the computation tree, unwinding all and-goals to the right of the new or-installation point.

Breaking orthogonality introduces serious overheads and complexities. First, unwinding parallel work is expensive. Second, it is waist-full. Imagine b and its sub-goals were

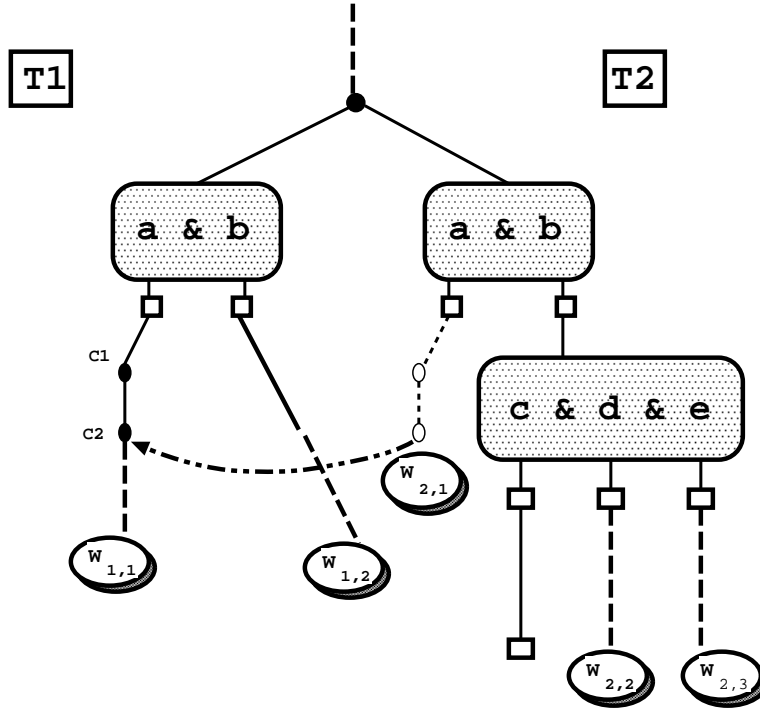


Figure 4.1: Epoch-Based And/Or Parallelism.

deterministic: we will still need to restart it after all work sharing operations. In fact, it may result in performing much more computations than in sequential Prolog. In the example, a would never reach a solution before c_2 , so the backtracking for b would never happen. Last, sharing operations will become slower, as teams must synchronise and will thus take longer to move to take work. So, both ANDP and ORP eventually suffer.

We would like to avoid epochs in order to guarantee orthogonality. Unfortunately, we next demonstrate that both PBA and ACE break orthogonality and may need epochs for correct execution. To do so we first study the C-tree in some more detail, and then we present the concept of *memory independence*, and why this is a necessary condition in order to obtain orthogonality. Armed with these concepts, we demonstrate how PBA and ACE break orthogonality.

4.2 Sharing Work in the C-Tree.

C-tree based models only or-share complete work. This guarantees that a single and-agent may be installing or-work per team. Moreover, this and-agent must have

backtracked into the public part of the tree, which is a connected tree of choice-points, rooted at the same root as the whole search tree. Thus this worker will be working as sequential Prolog would. All other workers will be working in *advance*, as it is expected in ANDP systems. The C-tree hence allows for orthogonality.

Guaranteeing that the public tree is connected can be implemented in a straightforward fashion. We just ensure that all newly published or-work is a contiguous sequence of choice-points, rooted in the shared tree. We thus obtain the topology of the shared tree. Note that implementations of IAP must also detect whether a sequence of choice-points is connected, so the C-tree does not require further machinery.

There is, however, one case where IAP can affect public or-work. The problem arises because shared choice-points may be discarded after semi-intelligent backtracking [75]. This is illustrated in figure 4.2.

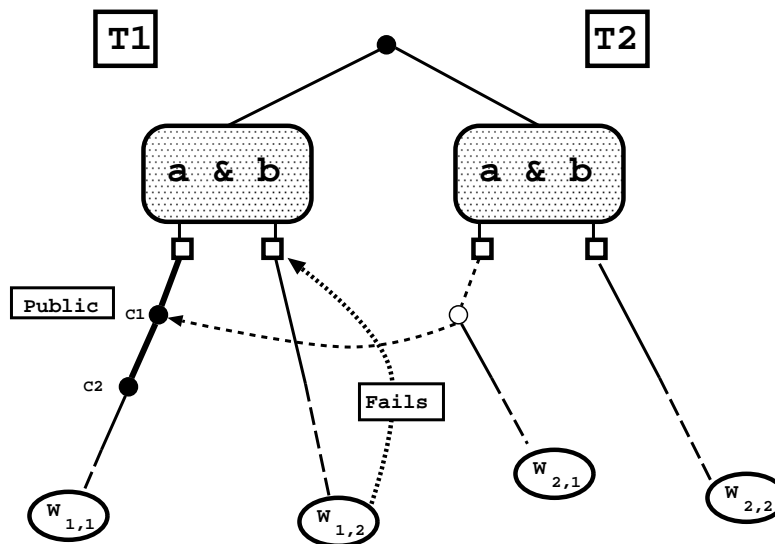


Figure 4.2: Inside killing of a public or-branch.

In figure 4.2 team T_2 took or-work from the choice-point c_1 from the public or-branch created by team T_1 when computing goal a . When installing the corresponding or-work, goal b is relaunched and taken by and-agent $w_{2,2}$. Meanwhile, the computation for b by $w_{1,2}$ fails while the parcall (a, b) is still in “inside” mode. Team T_1 must discard the parcall (a, b) , and so must team T_2 .

Moreover, if another team, say T_3 , was also searching for public or-work under choicepoint c_1 , the team should backtrack and move above the parcall (a, b) . In fact, we must discard (a, b) and all computations below. In terms of ORP, we say that team T_1 has implemented a $!$, fail operation up to the first choice-point above (a, b) . From

a different view-point, this is a new form of *intelligent backtracking* such that internal failure in a team can force other teams to backtrack and fail.

4.3 Memory Independence

Sharing of branches in And/Or systems is most often implemented by sharing a set of objects that represent executed branches. There is an alternative: we can share execution *paths*, as shown by the Delphi system [5]. In this case the system will actually re-execute the branches that it is logically sharing. But the costs of re-execution are quite high, and most often systems will share stacks.

The objects we share will be Prolog terms, but also parcalls, choice-points, and environments. Shared objects thus must have *public* values. Depending on the model, they may or not have *private* values. Each shared object must have a *key* that will be used to access it. Most often, the key is the object's address. The key must be able to distinguish the object from all other objects. Private values are used in Binding Array models. These systems actually maintain extra storage to preserve private values. A public function, whose value is independent from the worker, maps from the public value-cell to the location of the private value. In the traditional BA the function is often generated from looking at the contents of the value cell itself.

It is important to note that in ORP copying systems, such as Muse, different objects may have the same key at the same time. For instance, different variables have the same virtual memory address. In practice, this is not a problem because two objects with the same key must belong to different branches, and thus will never be live in the same computation.

There is an important symmetry with binding-array based systems such as Aurora. In this case, every object is guaranteed to have an unique key. But the public function may have the same value for two different objects. Again, two different cells in the same branches will not map to the same private value, because we guarantee that two objects that map to the same value must belong to different branches.

In both cases, we can only copy between or-agents when we guarantee that there is no such overlap. We do so by backtracking to a common node when we share. By backtracking we clean up data structures that could conflict: the stacks in the case of copying, and the Binding Array in the case of the SRI model. Is the same possible in the presence of IAP?

4.4 Sharing in the C-Tree.

As we have seen, the C-Tree model provides a high level design on how to combine ANDP with ORP. ACE and PBA provide an instantiation of the design with specific data-structures. That is, we need to guarantee that shared keys and private maps will not overlap between the teams. We name this concept memory independence:

Definition 3 Two teams are *memory independent* when we guarantee that **(i)** different objects always have different shared keys, and **(ii)** different objects always map to different private entries.

Intuitively, we say that two teams are memory independent if memory allocation within a team can *never conflict* with memory allocation within another team. We will show that guaranteeing memory independence severely restricts how we can share memory in both the PBA and ACE.

4.4.1 Memory Independence in the PBA

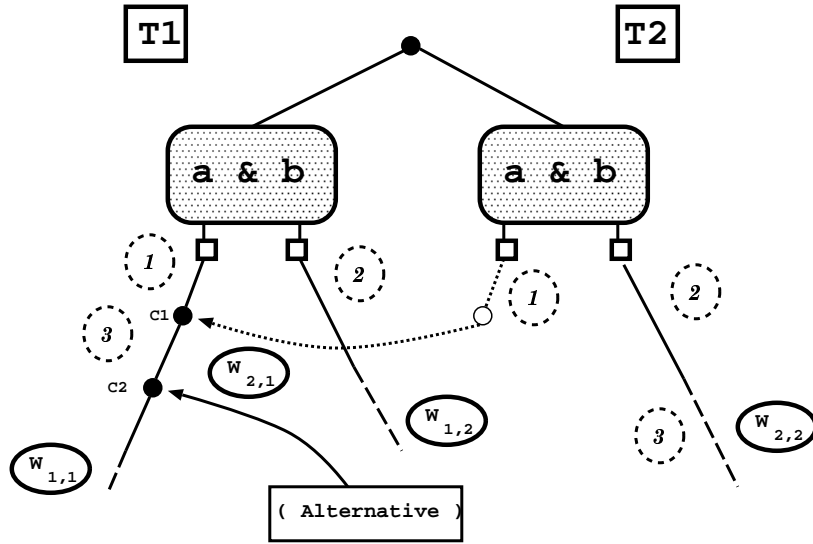


Figure 4.3: Structural problem in PBA.

We first study whether the PBA guarantees memory independence. We show this is not the case by using a counter-example. Consider the situation shown in Figure 4.3. The first team, (T_1) consists of two workers, $W_{1,1}$ and $W_{1,2}$. T_1 was the first team to exploit work from the parallel conjunction $(a \ \& \ b)$. Worker $W_{1,1}$ took the goal a and

created the choice-points C_1 and C_2 , whereas worker $W_{1,2}$ took the goal **b**. Observe that the dotted circles indicate how paged tables offsets have been allocated. Initially $W_{1,1}$ allocates page 1, and $W_{1,2}$ allocates page 2. Suppose now that between creating choice-points C_1 and C_2 , $W_{1,1}$ asks for a new binding-array page and receives page 3.

Execution for team T_2 proceeds differently. The team also has two workers. Worker $W_{2,1}$ first shares choice-point C_1 from T_1 whereas $W_{2,2}$ restarts execution of **b**. In this case, $W_{2,2}$ requires more variables more quickly than $W_{2,1}$ and asks for a new page receiving page 3.

At this point $W_{2,1}$ backtracks to node C_1 and the or-scheduler decides to take or-work from the closest node, in this case C_2 . Notice that from node C_2 to C_1 , T_1 used page table entry 3 and that this page is being used at the same time by $W_{2,2}$ in its computation of goal **b**.

As a result to move to node C_2 , team T_2 must rewind its computation of **b** in order to be able to install page 3 without conflicts. Moreover the problem may affect not just active computations, but in fact any previous independent and-parallel computations that were performed in advance and are already successfully completed. This constitutes work we would like not to rewind in spite of the fact that it is speculative, as it is not yet in the equivalent sequential search-tree.

4.4.2 Memory Independence in ACE.

The copying scheme used in ACE, described in section 3.3, suffers from a symmetrical memory allocation drawback to the PBA. This scheme, as proposed, is thus not able to guarantee memory independence between teams.

To illustrate why, consider the situation shown in Figure 4.4. As in the the previous case, team T_1 was the first to exploit work from the parallel conjunction (**a** & **b**). Worker $W_{1,1}$ starts by taking the goal **a** and creates the choice point C_1 . After a while, node C_1 is made public and team T_2 comes and takes work from this node, relaunching goal **b**. $W_{2,1}$ takes goal **a** and $W_{2,2}$ takes goal **b** for execution. Meanwhile, worker $W_{1,1}$ starts parallel execution of (**c** & **d** & **e**) and takes goal **c** for execution. Worker $W_{1,2}$ moves in and takes goal **d** for execution, eventually creating choice point C_2 in the process. Computation for goal **c** succeeds quickly and as a result node C_2 is made public. After goal **c** succeeds, $W_{1,1}$ takes goal **e** for execution. At this point the situation depicted in Figure 4.4 arises. Worker $W_{2,1}$ has failed and performs *inside*

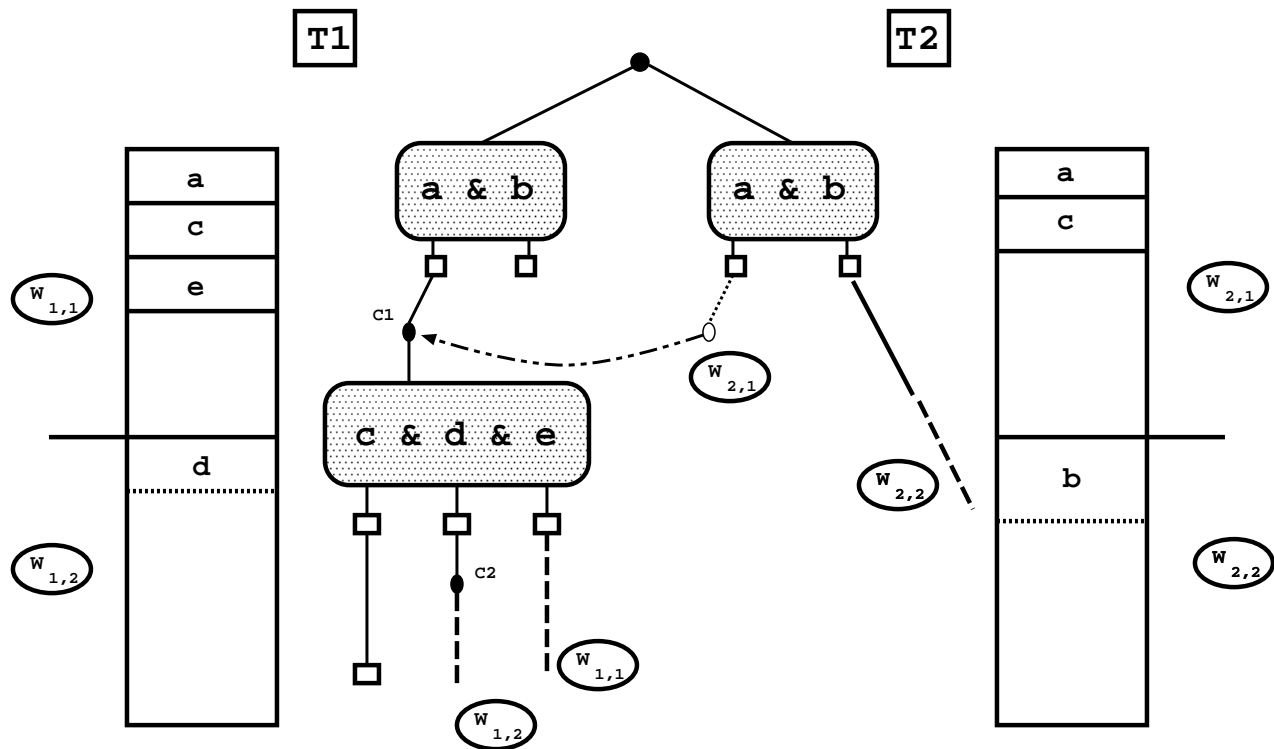


Figure 4.4: Structural problem in ACE.

backtracking to node C_1 . Since this is a public node the or-scheduler is called and it decides for $W_{2,1}$ to install work from the now public node C_2 .

The problem in ACE appears when worker $W_{2,1}$ in team T_2 tries to install work from C_2 . The space used for goal **d** is the space that team T_2 reserved for the execution of goal **b**. As a result worker $W_{2,2}$ cannot directly copy goal **d** to the stack of team T_2 . For the execution to continue T_2 has to rewind and deallocate the computation of **b** before the installation of or-work from node C_2 can proceed. Observe further, as in the PBA case, that in general the problem may not just affect close active computations, but any previous independent and-parallel computations that were or are being performed successfully in advance up to the root of the C-tree. Since these constitutes proven useful work they should not have to be rewinded.

4.4.3 Deterministically-Pure Reuse.

We also remark that memory dependency also makes it impossible to perform a further optimisation, the reuse of purely deterministic goals. The idea is that we would like to reuse pure computations completed in other teams when installing work. To illustrate

this concept consider figure 4.5.

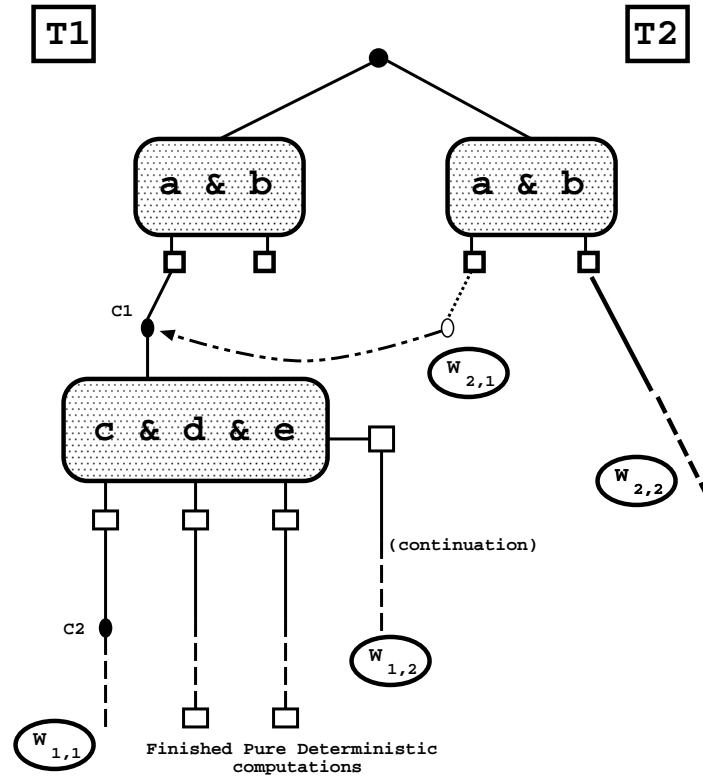


Figure 4.5: Deterministically-pure reuse policy.

In this situation, similar to the one already described to show that ACE suffers from the mandatory rewinding problem, we can observe that in team T_1 , the goals d and e of parcall $(c|d|e)$ have terminated in a deterministically pure state. By this we mean that no choice points have been created and no side effects performed during the goals execution. Notice that the computation of any of these goals could entail an arbitrarily large number of nested parcalls, each one of them containing completed pure deterministic and-goals. If any one of those sub-goals created just one choice-point or executed just one side effect the entire sub-branch would be tainted, making goal c or goal d non deterministically pure.

Observe that worker $w_{2,1}$ of team T_2 is going to install work from choice point c_2 . Without this optimization this would entail the recomputation of goals d and e on team T_2 . But since they are deterministically pure they are executed in exactly the same way no matter how many times, and no matter in which team they are executed inside the parcall $(c|d|e)$. They can thus be computed once and reused whenever needed inside this same parcall.

To achieve this, instead of relaunching goals d and e , worker $w_{2,1}$ of team T_2 installs the goal's bindings (in the case of a binding array based system) or copies the stacks (in the case of a copying approach) of these pure goals to its team.

In systems that do not guarantee memory independence the stacks for goals d and e in team T_1 can conflict with the stacks for goal b in team T_2 , goal b (and any other goal in the same circumstances in team T_2) has to be rewinded before this optimization could be applied.

4.4.4 Detecting the source of conflict

An alternative to epochs would be to solve memory conflicts simply by detecting which computations are the source of trouble and only rewind those, leaving the others as is. We argue that even if possible, this detection is very complex and costly. First, we still need to unwind independent computations.

Second, we observe that in order to detect which goals are affected, we need to maintain, for each goal, a list of all allocated pages (in PBA) or a list of all memory chunks (in ACE) used during computation. Then, and before it installs or-work, a team has to stop all computations first. Otherwise, more memory resources might be allocated. We thus must suspend all and-goals being currently computed. When installing or-work and going down along the branch, we must consult the lists each time a new page (PBA) or a new memory chunk(ACE) is reached. If we detect a conflict, then we must mark the source for rewinding. The conflict might involve an active worker, or completed work.

In the best case, we will have more complexity and benefit little. We thus argue that this does not constitute a viable solution to this problem.

4.4.5 Towards Memory Independence

We have shown that in both the PBA and ACE it is quite straightforward to produce simple examples where teams conflict in their runtime memory allocation behavior. More complex situations can and do arise.

In general, correct execution using the already proposed data schemes for environment representations requires a team to kill and fully rewind at least some of the teams independent-and private computations before installation of public or-work is allowed

to proceed. Therefore, we need to devise a data structure for environment representation that does not impose arbitrary restrictions in the way a team is allowed to share or-work and combine in an efficient way the solutions generated by independent and-computations.

One could extend the PBA and make it have a centralized and synchronized page management scheme, avoiding conflicts and guaranteeing maximum flexibility. However the cost of extra overheads caused by a centralized synchronization point imposes an unnecessary source of sequentializations in the system. Furthermore the extra overhead in dereferencing variables incurred by the “paging” scheme associated with the fact that globalizing page management to the system level (all teams would share a single system page offsets allocation table) would result in the existence of many “holes” in each of the teams private page table. The problems in this approach led us to research a more elegant and simpler solution.

A second alternative would be to use hash tables, as originally proposed for ECRC’s Pepsys [12] and then used by Shen for Prometheus [123] and later in Fire [125]. Because hash tables are allocated in shared space, they trivially guarantee memory independence. Unfortunately hash tables do not guarantee constant time binding access. This can become a problem in the presence of ANDP. In pure ORP systems they do guarantee constant-time task switching, but this is not possible in And/Or parallel systems because of the need to reinitialize and-work. Moreover hash-table management in an And/Or parallel system is even more complex than for ORP [125].

We would like a solution that does allow support for And/Or parallelism, at a minimum overhead over state of the art pure ORP and IAP systems.

4.5 The Sparse Binding Array

The *Sparse Binding Array* (hereafter SBA), takes its inspiration from the original binding array used in SRI based models. The key idea is that we can guarantee memory independence for variable bindings if the function that maps public to private cells is one-to-one. That is, we guarantee memory independence if every two variables, even in different branches, map to different private slots. We thus guarantee no conflicts. Furthermore, if several and-workers within a team create variables at the same time, they are guaranteed never to use the same memory cell for the different variables. To do so, the SBA simply creates a shadow of the whole execution stack for each team.

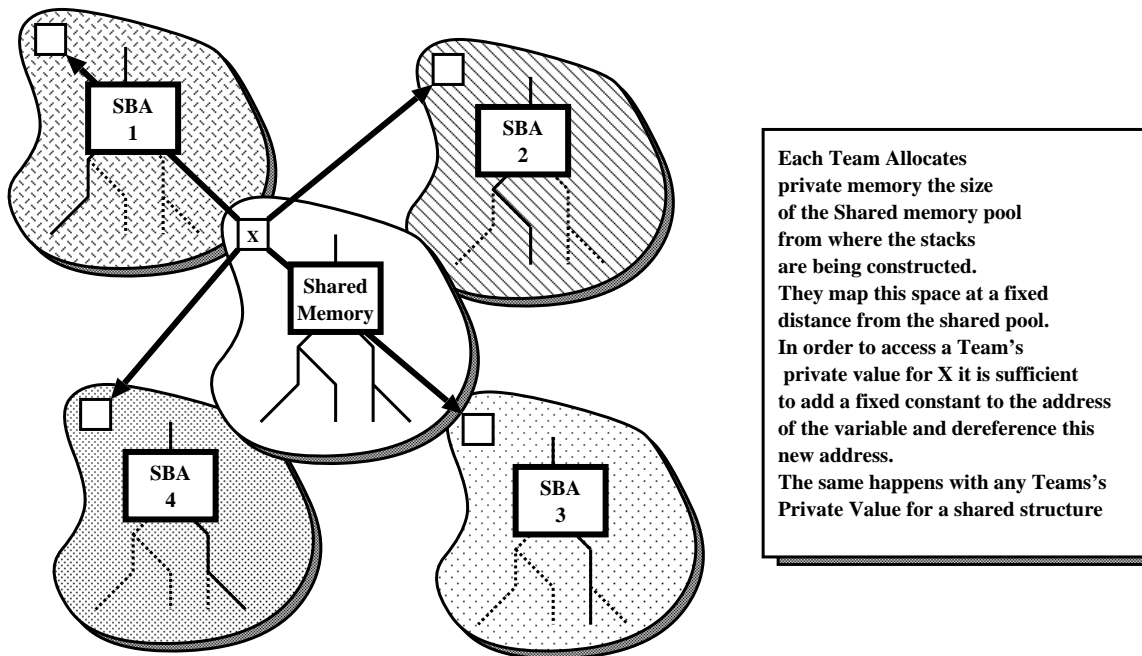


Figure 4.6: The Sparse binding array.

In a shared-memory shared-stack based parallel logic system, memory objects are built with blocks taken from a common pool of shared memory. This shared address space is then used to build each worker's stacks. Variables, among other system objects, are located in these stacks. To implement our binding representation scheme, the SBA, each team allocates a private virtual address space with the same size as the system's common shared memory pool. This local address space, is named the *sparse binding array*, and is mapped at a fixed virtual address location by each team in the system. Bindings that are private to a team will be stored in the local space, whereas unconditional bindings can still be placed in the shared data stacks. We can say that the SBA *shadows* the system shared address space.

In order to obtain a private values, SBA based systems can use a direct mapping from the shared memory pool to the team's SBA. The mapping is from addresses in the shared address space to addresses in the team's SBA and is as follows:

$$SHARED_PTR \Rightarrow SHARED_PTR + (SBA_BASE - STACK_BASE)$$

where *STACK_BASE* is the base address of the system shared memory pool, and *SBA_BASE* is the system wide base address for the SBA. As most operating systems allow memory mapping at user-defined addresses we can guarantee the difference

SBA_BASE – *STACK_BASE* to be a compile-time constant, and identical for all the teams of the system. In fact we can even specify a constant that is easier to add to the variables (or other system object) address. This is illustrated in Figure 4.6. In these systems, the private value of a variable can be obtained as follows:

$$PRIVATE_VALUE = SHARED_PTR[FIXED_DISTANCE]$$

where *FIXED_DISTANCE* is the constant value known at compile-time. Variable binding access is therefore a very fast constant-time operation. In fact it is faster than the same operation for the traditional binding array because we only need to add a constant to a pointer followed by a dereference operation. In contrast, the traditional BA needs to read the valuecell, add the read value to a base pointer and dereference again to obtain the variable binding. In our experiments in determining the practical applicability of the SBA concept, just for or-parallelism, this increase in access speed for variables bindings together with a better management for variable age, to be described later (section 5.4), is responsible for a 10% increase in the absolute execution speed of the Aurora system with just one worker [115].

4.5.1 Comparing the SBA with other systems

One motivation behind the design of the SBA was to take advantage of the constant improvement in operating system support for memory allocation, namely the advance in support for direct mapping of files in shared memory [145]. Modern operating systems allocate page frames lazily, that is, a new page is allocated only when first referenced. Consequently, we may request a big chunk of virtual memory, but only allocate much fewer real system resources, that is, page-frames and supporting swap space. In these category we find operating systems of the Unix variety like 4.4BSD, Solaris2.6, Digital Unix, and Linux. In fact, even in the cases where the Operating System does not fully support lazy memory allocation, we can implement the memory allocation ourselves through writing our own segment violation handler. A similar technique is used in the implementation of Distributed Shared Memory systems [108]. This means that in practice, we can always guarantee that the actual amount of physical memory *we need is not what we ask for, but what we use*.

To evaluate SBA memory usage we must therefore understand how much more memory we end up using relative to what we use in other environment representation schemes. To answer this question, we should observe that in an ORP “run” we need to use pages containing variables. Our worst case is therefore the case where all private

pages shadowing the pages containing variables would need to be accessed. Overhead in relation to a traditional BA scheme will be difference between the number of pages containing variables times page size and number of variables times variable size. If we accept that most pages contain variables, then the overhead as compared to the BA will be essentially total stack size minus cells initialised to be variables, that is, the number of cells not initialised to be variables.

In practice, recent implementations of the SBA only use the SBA for cells in the shared region. So the overhead is actually the numbers of cells not initialised to be variables in the shared area. We have analysed pure ORP systems [115] and observed that most cells in the environment stack are variable cells, and also that a large percentage of heap cells are also variables. Thus we expected the number of system memory pages really referenced (thus used) by a SBA based system to stay on the same order of magnitude than in the corresponding BA approach. Our results confirm this, and in fact the SBA-based system also has better execution times than a BA-based system, at least in the or-parallel case.

We believe that our results also stem the fact that we are using the WAM [151] as the underlying execution engine. The WAM has very good memory locality properties [141] and this reflects in the memory behavior of the SBA. Observe that if variables are located in the address range $A \dots B$, in the SBA we would have the corresponding ranges $A \dots B$ and $SA \dots SB$, so that in the worst case locality will be in the worst case half of what it was in the WAM.

To summarize, in the SBA, each cell in the local and global stacks is “virtually shadowed” as many times as there are teams. We use the word “*virtually*” because we are referring to virtual memory, managed by the operating system. Most of this memory will never be used during execution and will just be kept as reserved backing store.

4.5.2 System objects private values

Shadowing the whole stacks and not just variable cells has an important advantage in terms of simplifying system implementation: *Any* object whose memory has been allocated from the initial common shared memory pool can have a private value in the SBA in exactly the same way a variable’s private value has.

The SBA readily provides private values for team’s objects such as parcalls, choice-

points and computation-slots. To obtain the private value, one just needs to use the address of the object, add to this address a constant offset (the distance between the SBA and the initial address of the common memory pool) and dereference this pointer to access the team private contents for the object.

As we will see later, this property is very useful for our IAP/ORP system design and is fundamental for implementing the associated data structures. This proves to be extremely useful by simplifying system implementation.

4.6 Summary.

One of the major challenges in current logic programming research is how to efficiently exploit maximum parallelism in logic programs. We have presented an environment representation scheme that allows for the combination of ORP with IAP. We believe that our scheme will also perform well with dependent and-parallelism (DAP).

We showed that in order to obtain orthogonality between ANDP and ORP some reuse of goals between search branches is required. We have shown that this is severely restricted in previous environment representations schemes. In section 3.2.1 we have demonstrated that there are important shortcomings in PBA related to the way the page table is managed and in section 3.3 we saw how in ACE trail management and copying, in its current form, are too inefficient to be part of a successful implementation of a combined And/Or system.

This situation lead us to propose a new data structure, the Sparse Binding Array, that does not suffer from these shortcomings and at the same time simplifies the binding array management. Arguably, SBA-based approaches solve the problem of memory allocation within teams, while at the same time guaranteeing faster access to variable bindings. Further notice that in the SBA the value inside the variable cell is no longer needed to find the associated binding array slot. We will see that this value can however be used for variable age maintenance (section 5.4).

We will see in Chapter 6 that by using this scheme the main data-structures for both or-parallelism and and-parallelism can be naturally integrated in order to support our goal of designing and implementing a simpler IAP/ORP Prolog system.

Chapter 5

Or-Parallelism and the SBA

As a first step towards a combined SBA based IAP/ORP Prolog system and also as a concept validation scheme experiment, we took the Aurora [91] or-parallel Prolog system and converted its or-parallel binding representation scheme, a classic binding array, to the SBA. We concluded that the necessary system changes were reasonably small and very localized while the modified system performed well when compared to the original Aurora system.

In this chapter, we give a more in-depth discussion of the Aurora or-parallel system and of the changes we made to make it run with the SBA.

5.1 SBA Memory Allocation

In Aurora each stack consists of a doubly linked list of memory blocks. When a stack overflow occurs, a new block is simply allocated and linked at the end of the list and the relevant top of stack pointer is set to point to the base of the new block. To avoid fragmentation problems, progressively larger blocks are created (currently, each new block is allocated twice as big as the previous block). These blocks are never deallocated during run-time becoming a permanent part of a worker's stack. These memory blocks are allocated in increasing memory address order.

We have preserved Aurora's memory allocation scheme for the shared stacks but investigated two different approaches for the SBA implementation, namely:

MMAP: a large chunk of memory was allocated for the SBA. Unfortunately our

experience showed that, in a SPARC machine running Solaris2.4, the `MMAP` routines had severe problems. We detected wide variations of performance between runs. `MMAP` best performance was obtained when we actually allocated all the memory first. If we did not ask for the memory immediately, there would be a severe degradation in performance.

MALLOC: the alternative was to use `malloc` to allocate space for the binding array. Surprisingly this gave much more stable performance in our platform, and has since been used in our prototype implementation.

SHMGET: In Linux `MMAP` has some problems, specially if the flag `MAP_SHARED` is used. Our shared memory allocation routines are thus based on the SysV memory allocation routines of the IPC variety, namely `shmget`, `shmat` and `shmctl`. Later we switched our implementation of and-agents from `processes` to `threads` and the need for shared memory facilities in the SBA were no longer needed. As result currently our prototype implements the SBA with a simple `malloc`.

We believe that the problems in behavior with `MMAP` in the SPARC was related to the range of virtual memory addresses used by `MMAP` in the solaris2.4 operating system. The memory area used for `MMAP` had problems in being properly cached by the machine. If another system was used we believe the problem would disappear.

These side notes are simply the historical reason of why the SBA had always been subsequently implemented using `malloc`. Even when teams are later introduced and implemented, we still use `malloc` because and-agents are, in our system, implemented as threads inside the same process, sharing the SBA. Having each SBA associated with an operating system process allows for the required memory space to be dynamically allocated in the simplest and most portable way by just using `malloc`.

5.2 The Binding Array in Aurora

In the SRI model, conditional variable bindings are stored in the worker's private binding array to allow for constant-time access, and in the trail to allow workers to migrate from one branch to another in the search tree by maintaining a one-to-one correspondence between binding array contents and the variables that have been conditionally bound between the root node and the worker's current position.

As mentioned in section 2.7.3, every variable in the SRI model is associated with a unique variable number. In Aurora the number is positive in the case of a local variable and negative in the case of a global variable. A base register `binding_base` points at the array element corresponding to variable number 0. A BA element contains 0 iff the corresponding variable is unbound or if that element is outside the part of the array which is currently in use.

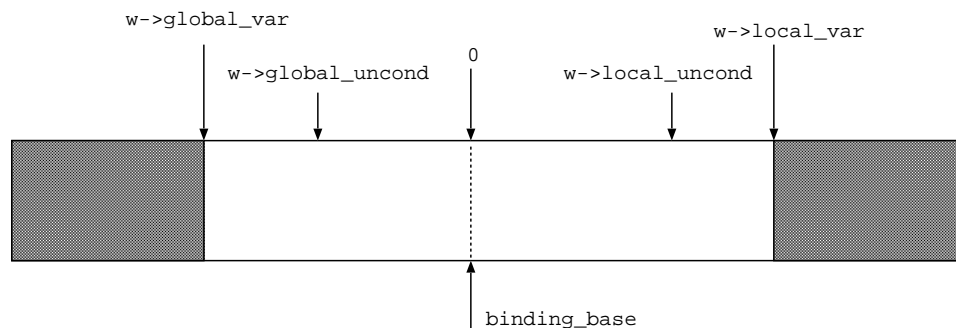


Figure 5.1: The binding array in Aurora.

Figure 5.1 depicts the binding array as implemented in Aurora. The range of variable numbers currently in use are defined by the two registers `w->global_var` and `w->local_var`. The subrange of variable numbers for variables which need to be bound conditionally are defined by the registers `w->global_uncond` and `w->local_uncond`. Grey areas correspond to unused areas and are filled with zeroes.

An advantage of this design is a better utilisation of the available memory space. If one part of the binding array overflows, more space can be provided by shifting the base register and array contents towards the part which is not yet full. If both parts fill up, a new area must be allocated, the old binding array contents copied and the `binding_base` set to point to the appropriate address in this new memory area.

5.3 The SBA in Aurora

In our implementation, the allocation of the SBA involves reserving virtual memory enough to “shadow” the whole of the shared stacks. This includes code space, choice-point stack, and trail which does not have to be shadowed. In practice it should not matter, as the space is only *virtually* allocated.

In any case, we can keep the offset `BA_BASE - STACK_BASE` fixed and known at compile-time. On the contrary in order to save space and dynamically maintain its

BA, Aurora does not have a fixed *BA_BASE* value. For more details please consult Carlsson [25]. This results in worse performance versus the SBA system, were the complete SBA is allocated at a fixed location from the beginning of execution.

5.4 Age and variable representation in the SBA

Variable age is the mechanism by which logic programming systems maintain creation order for variables. Ordering of variables is important for two fundamental reasons :

Conditional test — In order to decide whether a variable binding has to be trailed one has to determine if it is conditional. This is achieved by comparing the age of the variable with the age of the last created choice-point or parcall-frame, whichever is younger. If the variable happens to be younger, then the binding is unconditional and is recorded directly on the variable cell in the stack. Otherwise the binding is recorded in the corresponding cell in the SBA, because it is conditional. Conditional bindings are always trailed.

To understand why we need to consider the last parcall-frame in order to determine if a variable binding is conditional, consider the situation illustrated in figure 5.2.

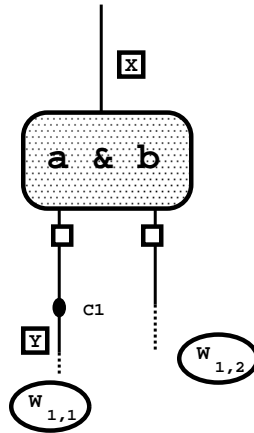


Figure 5.2: Conditional bindings under a parcall.

In the figure, variable *X* has been created before parcall-frame (*a|b*). Variable *Y* on the other hand has been created by worker $w_{1,1}$ after choice-point c_1 . Worker $w_{1,1}$ binds variable *Y* unconditionally. However when worker $w_{1,2}$ binds variable *X* the binding is conditional because the most recent parcall-frame (*a|b*) is younger than the variable.

If we did not consider the parcall-frame, the binding would be unconditional. If another team then came and installed work from choice-point c_1 it would find variable X unconditionally bound, which would be obviously wrong.

Order among variables — Secondly, when binding one variable to the other, one usually wants the newest variable to bind to the oldest. This guarantees that less conditional bindings are made and that no variable binding cycles among variables can occur. The exception is between local and global bindings as it is very important to guarantee that no global variable binds to a local variable, otherwise references from the heap to dangling locations in the stack could happen.

The question arises, if we use the SBA for environment representation, how to implement, in an efficient way, variable age in a IAP/ORP Prolog system. To answer this question let us first analyze our previous work on what should be the value of a variable cell within the SBA.

In the BA scheme a cell's value is an index into the binding array. This index is also used as the variable age. The PBA[64] generalises this scheme, but the the variable cell contents are still used to determine the location in the paged binding array of the variable binding. This value, not the variable address, is also used as the variable age in order to allow for maximum freedom in developing flexible memory management schemes.

One advantage of the SBA is that in order to find a variable binding, we only need to use the address of the variable cell. Its content is not used to this effect and as such we are free to use it to concentrate on obtaining a better design for variable age implementation.

We could have preserved the Aurora's method for variable age maintenance. Aurora has in fact two BA, and therefore two counters, one for local and other for global variables. The two counters are used so that environment trimming can recover local BA entries when the system recovers environment space. SBA based systems do not need this optimization, as last call optimization on the SBA follows automatically from last call optimization in the shadowed environment stack, so just one counter would be enough for the SBA.

A second and more interesting simplification is to avoid having a counter for every variable, and instead use a *variable level* representing the number of choice-points and parcall-frames above the variable cell in the execution tree. Observe that if we carefully

design our systems to always allocate memory in such a way that any private segment of a branch will always be composed of memory chunks in ascending order (as it is the case in Aurora) then variable age can be simply determined by the lexicographic order of the pair (**variable_level**, **variable_address**). This scheme has the advantage of not having to update a counter each time a variable is created as the *level* will be updated only when a choice-point or parcall-frame are allocated.

The new scheme involves less bookkeeping than Aurora, because we do not need to update BA counters whenever we create variables. On the other hand it can be more expensive when we need to compare variable ages, as in the worst case when we may have to perform two comparisons, one for the levels and another one for the addresses, if the levels prove to be the same. In fact, a single comparison is sufficient for most cases, as we shall see later.

We were very successful in using this scheme with our implementation of the SBA in the Aurora system, showing that “sometimes two comparisons” overhead is clearly supplanted by the extra benefit of not having to increment a counter each time a new variable is created.

Proper “level” management is complicated in the presence of a cut. When we perform a cut several choice-points can be discarded. If we are at level O , perform a cut to level M , and bind a variable created at level N , where $M < N < O$, in this order, this variable binding is unconditional. One naive solution to the problem would be to reset the “level” counter to M as soon as the cut is performed. Unfortunately the problem is that the next time we create a choice-point, its level would be $M + 1$. As a result all variables already created at levels between $[M + 1, O]$ would now bind unconditionally, when in fact they can only be bound conditionally.

Our solution to this problem is to use two level counters. The first counter, **level-top** represents the deepest level we have reached so far in the branch, whereas **level-uncond** represents the level for unconditional variables. Private operations with these level counters are thus as follows:

```
new-var : *CELL_ADRESS = level-top;
```

```
conditional : (*CELL_ADRESS < level-uncond)
```

```
try : B→level = level-uncond = level-top = level-top+1;
```

```
retry : level-top = level-uncond = B→level;
```

trust : `level-top` = $B \rightarrow \text{level}-1$; `level-uncond` = $B \rightarrow \text{parent} \rightarrow \text{level}$;

cut : `level-uncond` = $NB \rightarrow \text{level}$;

The **new-var** operation creates a new variable. After this operation the variable cell contains the current engine execution “level”.

The boolean **conditional** gives whether or not a variable binding is conditional. Observe that only one comparison is needed to determine whether a variable is conditional. The worst case of two comparisons only happens when comparing the age of two variables.

The **try** operation creates a new choice-point. This operation starts by incrementing the top-level, which also becomes the new level for unconditional bindings. It then sets both the unconditional and conditional levels to the new value.

The **retry** instruction recovers the two levels from the choice-point.

The **trust** instruction removes a choice-point from the top. Immediately after this operation and before a new choice-point is created, variables are created with the same level they were being created immediately before this choice-point was created. The `level-uncond` is set to the level of the parent’s choice-point. Notice that `level-top` and `level-uncond` may be different as a result of a previous **cut**.

Finally, note that **cut** sets `level-uncond` to the “cut choice-point” level but maintains `level-top` undisturbed.

5.5 The SBA engine in the Private Region

Most of the time the Aurora engine executes private work [136]. Execution in the SBA based system, henceforth simply SBA, follows exactly the Aurora behavior. The only difference resides on the implementation of the macros for variable access, variable age comparison and management of variable levels.

After introducing the “level” scheme, we had to make changes to other operations, such as choice-point manipulation and **cut**. These changes follow the principles presented in section 5.4. Observe that in order to support shallow backtracking [24] the actual WAM implementation in Aurora is slightly more complex than in a more traditional WAM. But this issue should be orthogonal to the SBA.

A small subtlety arises with the implementation of tail recursion in Aurora. Aurora keeps a free variable in every environment in order to calculate its age. To recover space it tests whether this age is younger than the age for the last choicepoint. The SBA implementation uses a similar approach, giving to each environment an extra slot with its age. Environments can thus be recovered if they are younger or if they have the same age as the current `level-uncond`.

There is however one small inefficiency in the current SBA implementation, inherited from Aurora. This occurs because a worker always accesses a variable binding array entry whenever it dereferences the variable. These accesses are unnecessary if the variable has been created after the last choice-point, and could therefore be avoided because the variable is necessarily private.

Whenever it creates a choice-point, Aurora also verifies if there is an overflow in the binding array. This is not required for the SBA, thus accelerating choice-point creation.

5.6 Parallel Work

Parallel Work happens in Aurora when the engine enters the shared area of the search-tree, or when the engine calls the *or-scheduler*.

Work in the shared area can be: *moving up* the tree to backtrack, *moving down* the tree to select new work, and *cutting shared work*. Using the SBA does not enforce any changes to this scheme except that it is necessary to reset “levels” to appropriate values whenever we backtrack to shared areas of the tree, or whenever we create new work. Note that when creating work we never perform `trust` in the shared area because we cannot be sure someone else is exploring an alternative. The choice-point has been created before, hence the operation is always a `retry`. In Aurora the `retry` is always followed by creating an embryonic choice-point. These rules had to be followed very carefully for the new system to run properly.

5.7 Initial Performance Evaluation

We have measured the timings and speedups obtained both from Aurora with SBA and Aurora with binding arrays (table 5.1 and table 5.2). We have used an 8-CPU

SparcCenter 2000 with 256MB of main memory, 1MB cache per CPU, running Solaris 2.4. The system was in multi-user mode. Both versions of Aurora were compiled with `gcc -O2`. We used the standard Aurora benchmark set. The scheduler we use is the distribution release of the Bristol scheduler [14], as ported to the SPARC [114].

With one worker the SBA is between 10 and 15% faster than Aurora. Increasing the number of workers, the SBA generally outperforms Aurora, although for higher number of workers Aurora comes close and sometimes actually performs better than the SBA.

We believe that the better sequential performance is explained by the inherent advantages of the SBA: easy calculation of the BA cell address and the use of levels for age comparison of variables.

As regards speedups, some of the reasons we think why Aurora can achieve better numbers are:

- Aurora has worst one-worker performance, and therefore has coarser task granularity. This means that scheduling overheads will be less significant.
- Task-switching in the SBA is more expensive because the page working set is bigger.

However, we expect the speedups to improve by using the newer releases of the Bristol scheduler. Finally note that these results are still preliminary. Further tests and optimizations will be carried out.

5.8 Performance comparison with other systems

The literature includes several comparisons of copying-based versus BA-based systems, and particularly of Aurora vs. Muse [14, 38]. One problem with these studies is that Aurora and Muse have very different implementations: it is quite difficult to know whether the differences stem from the model or from the actual implementation.

In contrast, our group compared three different forms of environment representation, Copying (as in Muse), *alphaCOW* and SBA, by implementing them over the same YapOr system [110]. The system is derived from the Yap engine [112]. This is one of the fastest emulator-based Prolog systems currently available, and only 2 to 3 times

Goals [*Times]	num_workers						
	1	2	3	4	5	6	7
parse1*200	1.62	1.14(1.42)	1.13(1.44)	1.25(1.30)	1.25(1.30)	1.26(1.29)	1.27(1.27)
parse2*200	6.36	3.64(1.74)	3.20(1.99)	2.83(2.24)	2.71(2.35)	2.61(2.43)	2.54(2.50)
parse3*200	1.32	1.03(1.29)	1.04(1.28)	1.13(1.17)	1.16(1.15)	1.20(1.11)	1.24(1.07)
parse4*50	5.81	2.97(1.96)	2.28(2.55)	1.88(3.09)	1.69(3.44)	1.55(3.76)	1.46(3.98)
parse5*10	3.86	2.04(1.89)	1.45(2.66)	1.20(3.23)	1.04(3.72)	0.92(4.20)	0.87(4.44)
db4*100	2.52	1.58(1.60)	1.19(2.12)	1.02(2.48)	0.92(2.73)	0.90(2.82)	0.86(2.94)
db5*100	3.27	2.03(1.61)	1.45(2.25)	1.23(2.65)	1.14(2.88)	1.07(3.06)	1.03(3.17)
farmer*1000	2.96	2.48(1.19)	3.09(0.96)	3.61(0.82)	3.96(0.75)	4.21(0.70)	4.51(0.66)
house*200	4.09	2.69(1.52)	2.38(1.72)	2.20(1.85)	2.10(1.94)	2.04(2.01)	1.98(2.06)
8-queens1*10	7.00	3.58(1.95)	2.44(2.87)	1.85(3.79)	1.52(4.61)	1.29(5.43)	1.13(6.18)
8-queens2*10	17.99	8.89(2.02)	5.98(3.01)	4.50(4.00)	3.70(4.86)	3.11(5.79)	N/A
tina*10	14.68	8.20(1.79)	5.48(2.68)	4.38(3.35)	3.62(4.05)	N/A	N/A
sm2*100	10.03	5.35(1.88)	4.05(2.48)	3.10(3.23)	2.63(3.81)	2.33(4.30)	2.10(4.77)
AVERAGE		(1.68)	(2.15)	(2.56)	(2.89)	(3.07)	(3.00)

Table 5.1: Performance of SBA on a SparcCenter 2000

Goals [*Times]	num_workers						
	1	2	3	4	5	6	7
parse1*200	1.86	1.17(1.59)	1.12(1.67)	1.18(1.58)	1.20(1.55)	1.21(1.54)	1.26(1.48)
parse2*200	7.00	3.78(1.85)	3.07(2.28)	2.73(2.56)	2.61(2.68)	2.56(2.73)	2.62(2.68)
parse3*200	1.49	1.10(1.36)	1.11(1.34)	1.18(1.26)	1.19(1.25)	1.24(1.20)	1.30(1.15)
parse4*50	6.07	3.25(1.87)	2.44(2.49)	1.96(3.10)	1.76(3.45)	1.63(3.72)	1.54(3.96)
parse5*10	3.99	2.13(1.87)	1.50(2.66)	1.24(3.22)	1.05(3.78)	0.94(4.22)	0.87(4.59)
db4*100	2.57	1.50(1.71)	1.15(2.24)	0.98(2.63)	0.89(2.90)	0.86(3.00)	0.85(3.05)
db5*100	3.15	1.82(1.73)	1.32(2.39)	1.18(2.68)	1.08(2.91)	1.04(3.04)	0.99(3.19)
farmer*1000	3.24	2.36(1.37)	3.07(1.06)	3.63(0.89)	3.95(0.82)	4.16(0.78)	4.49(0.72)
house*200	4.36	2.63(1.66)	2.37(1.84)	2.18(2.00)	2.09(2.09)	1.99(2.19)	1.97(2.21)
8-queens1*10	8.14	3.84(2.12)	2.59(3.15)	2.06(3.96)	1.66(4.90)	1.40(5.83)	1.19(6.84)
8-queens2*10	20.06	9.75(2.06)	6.37(3.15)	4.79(4.19)	3.89(5.16)	3.32(6.04)	2.92(6.88)
tina*10	15.57	7.95(1.96)	5.51(2.83)	4.21(3.70)	3.47(4.49)	2.94(5.30)	2.62(5.94)
sm2*100	10.24	5.42(1.89)	3.75(2.73)	3.03(3.38)	2.51(4.07)	2.24(4.58)	2.07(4.94)
AVERAGE		(1.77)	(2.29)	(2.70)	(3.08)	(3.40)	(3.66)

Table 5.2: Performance of Aurora on a SparcCenter 2000

slower than systems that generate native-code. Yap should be between 2 to 4 times faster than the sequential Aurora engine on the same hardware.

In order to compare the performance of these three models [116] the three different systems were run in two parallel architectures: a Sun SparcCenter2000 with 8 CPUs and 256MB of memory, running Solaris2.7, and a PC server with 4 PentiumPro CPUs/200MHz/256KB caches and 128MB of memory, running Linux2.2.5 from standard RedHat6.0. Each CPU in the PC server is about 4 times as fast as each CPU in the SparcCenter. All systems used the same compilation flags.

A standard set of all-solutions benchmarks, widely used to compare ORP logic programming systems [136] was used as a source of all-solutions benchmarks. This kind of benchmarks is not susceptible to speculative execution, and the goal was to compare the three models. The benchmarks include the n-queens problem, the puzzle and cubes problems from Evan Tick's book, an hamiltonian graph problem and a naïve sorting resolution. Table 5.3 shows the execution time, in seconds, for Yap Prolog and the overhead, in percentage over Yap Prolog, introduced by each or-parallel model when executing with one worker.

Programs	Yap Prolog		YapOr		α COWL		SBA	
	PC	Sparc	PC	Sparc	PC	Sparc	PC	Sparc
cubes5	0.216	0.753	2%	13%	4%	10%	9%	21%
cubes7	2.505	9.042	1%	7%	3%	5%	7%	17%
ham	0.435	1.537	4%	20%	4%	24%	25%	57%
nsort	34.810	142.161	1%	14%	1%	21%	22%	55%
puzzle	2.145	8.411	2%	22%	2%	18%	22%	39%
queens10	0.703	2.809	3%	8%	4%	7%	12%	17%
queens12	20.921	84.600	2%	4%	3%	4%	10%	15%
Average			2%	13%	3%	13%	15%	32%

Table 5.3: Overheads Yap Prolog/Or-Parallel Models with one worker.

The overhead for copying (YapOr) confirmed previous results, and is of the order of 2% or 13% on PC/Linux and Sparc/Solaris, respectively. The overhead obtained for α COWL is equivalent to copying. The results are consistent, and the variations are quite above the noise in our measures. We expected performance to be about the same, as for a single processor we execute quite the same code: the systems only differ in their scheduling code, and this is never activated.

The overhead for SBA is, as expected, higher but not very much so, only of 15% or 32%. In fact, SBA vs. YapOr performs relatively better than Aurora vs. Muse. These

are very good results that gives further support to the viability approach and gives credit for further research with the SBA.

Programs	2 workers			3 workers			4 workers		
	Copy	COW	SBA	Copy	COW	SBA	Copy	COW	SBA
cubes5	1.99	1.88	1.98	2.97	2.60	2.97	3.95	2.69	3.98
cubes7	1.99	1.98	1.99	2.99	2.92	2.99	3.99	3.83	3.98
ham	1.97	1.90	1.99	2.93	2.64	2.97	3.82	2.79	3.87
nsort	2.03	2.00	1.96	3.06	2.98	2.93	4.08	3.90	3.90
puzzle	1.97	1.93	1.95	2.96	2.41	2.92	3.94	3.20	3.88
queens10	1.99	1.88	1.99	2.96	2.12	2.97	3.92	2.42	3.92
queens12	2.00	1.99	1.99	3.00	2.86	2.99	4.00	3.82	3.98
Average	1.99	1.94	1.98	2.98	2.65	2.96	3.96	3.24	3.93

Table 5.4: Speedups for the three models on the PC Server.

Table 5.4 shows speedups for the PC Server, and Table 5.5 for the SparcCenter. **Copy** is used for copying and **COW** for the α COWL. The results show that the best speedups are obtained with copying. The SBA follows quite closely.

Programs	2 workers			4 workers			6 workers			8 workers		
	Copy	COW	SBA	Copy	COW	SBA	Copy	COW	SBA	Copy	COW	SBA
cubes5	2.00	1.83	1.96	3.95	2.72	3.70	5.79	2.87	4.88	7.35	2.32	6.02
cubes7	2.01	1.97	1.99	3.98	3.79	3.87	5.97	5.03	5.53	7.74	5.87	7.40
ham	1.98	1.78	1.90	3.79	2.54	3.98	5.57	3.00	5.33	6.97	2.15	7.29
nsort	1.94	1.97	2.02	3.83	3.77	4.01	5.69	5.42	5.88	7.42	6.03	7.77
puzzle	2.02	1.92	1.91	3.94	3.08	3.64	5.91	3.79	5.12	7.68	3.79	7.08
queens10	2.03	1.85	1.93	3.97	2.36	3.82	5.83	2.73	5.48	7.47	2.43	6.95
queens12	2.01	1.95	1.97	4.01	3.74	3.92	5.97	5.13	5.89	7.77	5.81	7.66
Average	2.00	1.90	1.95	3.92	3.14	3.85	5.82	4.00	5.44	7.49	4.06	7.14

Table 5.5: Speedups for the three models on the SparcCenter.

5.9 Summary

We conclude that the sparse binding array is an efficient environment representation scheme. It can also be easily ported into traditional binding array based systems. In this chapter we showed that for Aurora the ensuing or- implementation can actually outperform the corresponding binding array based implementation.

In the end the or-system proved to be sufficiently stable to run a wide series of benchmarks that we used to validate the SBA as a viable approach. Since we were

able to use the Aurora or-scheduler as is, without any change, we were able to obtain meaningful comparisons between the BA and SBA approach that solely reflect the relative performance of these two approaches, without being affected by a different system behaviour that would certainly arise if the or-schedulers were not the same.

More recent results were obtained when the SBA was implemented in the `YapOr` system and its performance compared with other concurrent approaches for or-parallel environment representation. These results confirmed the good results we had with our previous experiences with Aurora. The SBA is a valid alternative to copying, specially as an interesting alternative for the applications where copying does not work so well. In what follows we are going to see how we can take advantage of the SBA to implement combined IA/OR parallelism.

Chapter 6

Integrating IAP/ORP with the SBA

In this chapter we present how to use the SBA in order to combine the classical models for the exploitation of IAP, such as the P-WAM [74] and &ACE [105]) with a binding array based ORP system, such as Aurora [91].

We describe the major modifications and extensions to classical ORP and IAP systems that we believe should be made for a successful combination. We start by redesigning the most important IAP/ORP data structures. Throughout, we use the orthogonality principle as a guideline. We then proceed by giving some the main algorithms of the And/Or execution model.

We terminate the chapter arguing that the complexity of classical execution models for IAP seriously difficulties the implementation of a combined system.

6.1 Or-workers as teams of and-agents

In our IAP/ORP design, teams exploit or-work (section 3), and each team consists of and-agents doing iap-work. This organization of and-agents into teams has the following advantages:

- When and-agents fetch and-work they do not need to load their binding arrays, hash tables, or copy their new environment from other and-agents. For example, if the system is executing `(a & b),c`, any member of the team can take the

cge-continuation **c**.

- The team concept complies with the principle of orthogonality described previously. As such it allows the implementor to reuse code and ideas from previous implementations of other ANDP and ORP systems.
- Systems that are designed to support full Prolog need to know whether side-effects builtins can go ahead. The concept of a team allows us to clearly decompose *leftmostness* search into *and-leftmostness* and *or-leftmostness* searches [65].

6.2 The SBA team process model

A team is composed by two distinct parts:

1. An address space shared by several and-agents. This address space contains two parts. **(i)** the shared system memory pool, which is in fact shared by every team in the system; and **(ii)** the SBA, which is private to a team.
2. A set of and-agents (in the spirit of &-Prolog or &ACE)

The structure of teams naturally maps teams to processes and and-agents to operating system threads, or light-weight processes. A team is the *address space* of an operating system *process* that contains several *threads* performing and-parallel computations.

6.2.1 Migrating And-Agents

The major drawback of teams is that maximum ORP is given by the number of teams, and maximum ANDP by the maximum number of workers in a team. Unfortunately, the user may not be aware whether the application is mostly an ORP or an ANDP parallel one. Thus, a fixed distribution of workers by teams may result in sub-optimal speedups [48]

We would therefore like for and-agents to be able to migrate from team to team, depending on parallelism. Dutra has shown [47] that reconfiguration of teams works quite well for the Andorra-I And/Or system-I. The work-based and history-based policies proposed by Dutra should apply to any And/Or system.

One problem with Andorra-I is that reconfiguration had to be performed by explicitly moving a worker between teams. This was an artifact of the scheduling and memory allocation techniques used in the system. In contrast, we suggest that our Operating System analogy should be taken one step further. Instead of explicitly migrating threads, we propose that each team will have a pool of suspended threads (that is, not using *cpu* resources) that are made active whenever necessary. Whenever a team has excess of and-agents it will *yield* one of its processors to a different team by suspending one of its processor. To yield an and-agent, a *migration message* is sent to another team. Our idea is analogue to Anderson's well-known scheduler activations mechanism [6], used by Operating Systems such as Solaris and Irix.

Summarizing, each team contains an internal array of system threads. Some are doing and-work, others are executing and-scheduler code. The rest are suspended, waiting for migration messages to be sent by the other teams.

6.3 And-Agents versus Or-Parallelism

The life of an and-agent is shown in Figure 6.1. The white-background nodes correspond to purely and-work within a team. They are:

- **Forward** — The and-agent is performing forward execution.
- **Scheduling** — The and-agent looking for and-work to do.
- **Backtrack** — The and-agent is backtracking within the private part of the team.

In Figure 6.1 the shadowed area corresponds to situations where the and-agent must be aware of or-parallelism. They are:

- **Public Backtracking** — The and-agent is backtracking within the public part of the search tree.
- **Move Down** — The and-agent is installing public or-work form the search tree.

Note that the and-agent interacts with the rest of the team by sending messages or by publishing and-work.

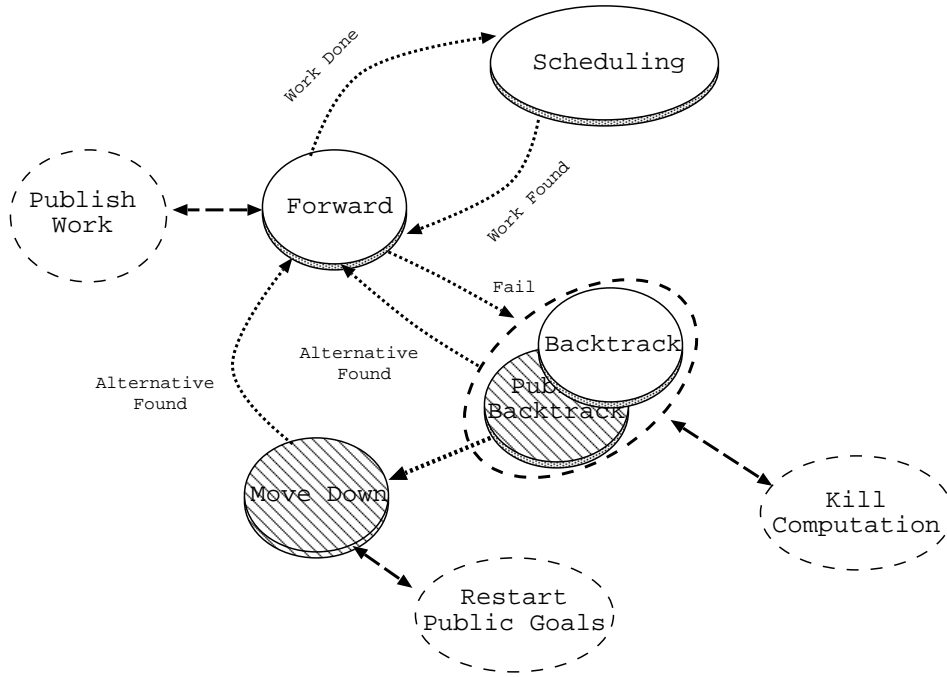


Figure 6.1: And-Agent Lifecycle.

Orthogonality means that when an and-agent is moving in an area shared with other teams the rest of the team will be working in and-parallel. Hence, they should be unaware of the fact that the and-agent is backtracking to and fetching work from the shared part of the search tree. Orthogonality also means that any and-agent may take or-parallel work.

Every and-agent must be able to interface with the or-parallel component of the system. Following the Aurora engine-scheduler interface [137] there are three basic interface operations that must be supported.

1. **MakeNodePublic** — make private work a part of the logically shared Or tree.
2. **MoveUP** — backtrack in the shared search tree.
3. **MoveDown** — an and-agent in a team moves to new alternatives in the shared search tree.

The **MakeNodePublic** routine shares a choice point node. Its tasks are essentially the same as in the Aurora system. The major task is to update choice-point fields used by the or-scheduler to inform that the corresponding or-work is now shared. The routine also has to navigate through the choice-point stack, in this case traversing the novel

data structures imposed by the presence of IAP. Remember that in order to facilitate IAP/ORP integration, we assume that only complete and contiguous and-parallel work is or-shared.

The `MoveUP` engine routine must also be made aware of IAP data objects and of the way IAP computation segments are physically and logically connected so that they can be reused after backtracking. As we explain next, moving up in the public And/Or tree involves decrementing the number of references of all the IAP/ORP objects found. We use a reference counter for each IAP/ORP object in the choice point stack: **(i)** each time a team moves down over one of these objects, its team reference counter is atomically incremented; **(ii)** When a team moves up over an IAP/ORP object this reference counter is atomically decremented; **(iii)** When the team reference counter reaches the value zero then the object is no longer referenced by any team in the system and can thus be logically deallocated.

By logically deallocating an object we mean marking the memory area that it uses as free. To actually release the area we need for it to be on top of the other, otherwise it effectively becomes a hole in the stacks. The and-agent who owns that piece of memory will be able to reuse it only during backtracking. This is the same principle as used in Aurora, proviso that we now have more data objects to take care of.

6.4 Or-Sharing And-parallel Data Structures

It should by now be clear that several teams may be sharing an or-public CGE. Each team may be executing in a quite different computational state: some may have terminated the CGE, others may be only be working on the leftmost branch. One major advantage of the SBA is that it trivially gives each team a private space where to store its own state. This is because in order to guarantee memory independence, the SBA shadows the teams shared address space in its entirety. All data structures, be they ORP related or IAP related, are allocated within the team's shared address space. Therefore, every data structures we use to control ANDP execution will also have an SBA shadow where private values can be saved.

An equivalent approach is to look at team-private values for IAP data structures as mutable conditional variables whose updates will be made directly into the its SBA position. Modifications will need to be properly *trailed*, not only for backtracking but also for proper or-work installation when other team installs or-work containing these

data structures.

6.4.1 Goal Computation slots

One important example of an IAP data structure that needs to be stored into the SBA when or-shared are the goal computation slots. Computation slots are directly updated in the heap, always while computation is or-private. Its value fields will move to the SBA, and updates will be made directly in the SBA when one of the goals for the same cge is made or-parallel.

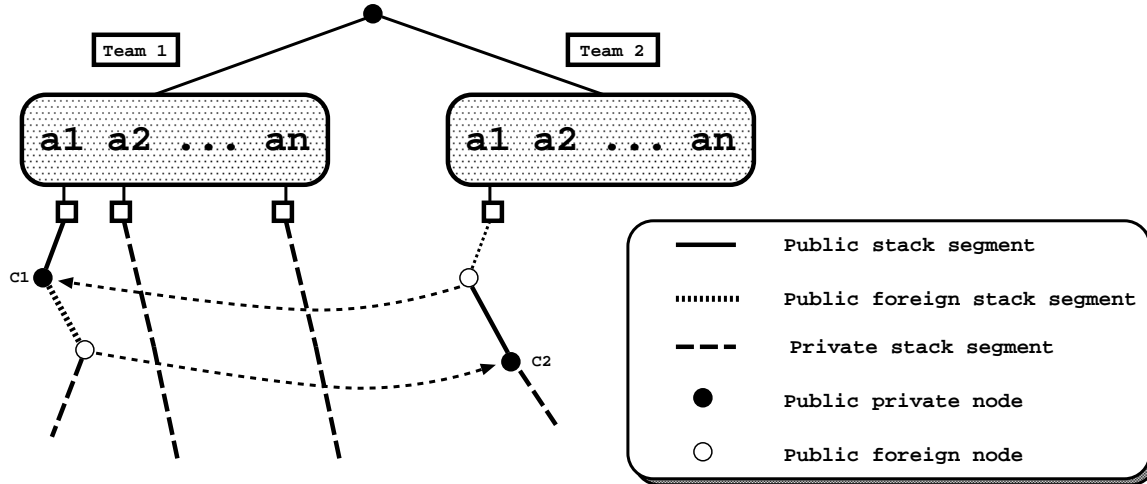


Figure 6.2: Or-public and or-private goals.

Notice that in the data-structure for CGEs we can have goals to the left of some goal g_i that are or-shared while goals to the right of g_i still remain private. Consult figure 6.2 for an example. Or-public and-data must always be updated in the SBA and any further change trailed, so that other teams may properly install the data.

The **MoveDown** routine is responsible for the installation of new or-work. The key idea here is that the **MoveDown** and-agent routine actually moves from the bottom to the top of the shared tree. Thus, the first time it finds a parallel conjunction it has to initialize the SBA parallel conjunction values, copy the state of the computations of the goals to the *left* and restart all and-parallel goals to the *right* of the current goal, allocating new computations slot data structures for the just restarted goals. All slots are shared and therefore data can be easily copied into the SBA and also available from *trail update entries*. And- slot values (the same happens with any other or-shared iap- data structures values) must be treated as a mutable variable [7] and therefore only the most recent trail update entries can be used to modify their values.

6.5 The Team Or-Representative

The Aurora system uses a special choice-point node, the *sentry* node, in order to separate the private branch from the or-shared part of the search tree. In our And/Or system we can still use or-sentry choice-point nodes to separate the And/Or private part of the computations (what is going on inside the team) from the or-shared public completed work (what has already been made public to the other teams).

ORP systems usually share or-work lazily and on demand. We would like to maintain the same policy for the integration of IAP/ORP. But since only completed and-work can be or-shared we have to devise an efficient method to **(i)** detect that the work we are or-sharing is in fact complete; **(ii)** share as much work as possible because this operation is in general more costly than in the pure or- case and should therefore be performed much less often.

In a classical or-parallel system a worker asks for or-work from another worker by sending a message requesting or-work to do. In the integrated system a team asks for work from another team by sending a message requesting or-work. Unfortunately the similarity must end here, because a team is composed by several and-agents working in and-parallel. We must therefore define which and-agent will be responsible to accept the message and publish or-work, and how much of the team's private or-work is going to be or-shared in the operation.

Our proposal is as follows. When a request for or-work arrives we start by assuming that all active members within the team are capable of publishing or-work (idle and-agents are thus excluded from this operation). Each of these agents tries to move up the or-tree and reach the *or-sentry node*. Complete independent stack segments are implicitly linked by IAP data structures in Prolog order and each active and-agent within the team follows this implicit link up into the and/or- tree until one of two things happens:

- The and-agent reaches a CGE and the goal immediately to the left of the and-agent current position has not terminated. In this case the and-agent cannot publish its work.
- The and-agent reaches the or-sentry node. Classical models guarantee that only one and-agent will in this way be leftmost, and thus able to reach the or-sentry node. Otherwise, there would exist two leftmost tasks, which is absurd. We can also guarantee that leftmost task will be an active task if we guarantee that a

parallel goal only starts after all goals to its left have started.

We name the team's and-agent that reached the or-sentry node the team *or-representative*.

The *or-representative* acts as a regular or-worker in deciding whether it can publicize or-work and how much of the or-work it is going to be made available. This is a pure or-scheduling decision and the integrated system can thus make use of the extensive research [15] that has already been made available on this subject.

The maximum amount of work that can be made or-public is the segment of the computation tree that starts at the current branch position of the team's *or-representative* and that terminates at the or-sentry node.

Since or-work installation should be in general more costly than in a pure or-parallel system we suggest the *share on the bottommost* policy for or-scheduling in the integrated parallel system, as this policy in general minimizes the number of or-work installation phases that will be performed throughout execution.

Backtracking into the Sentry Node

A team moves up within the shared or-tree when one of the team's and-agent backtracks into the or-sentry node. Since only complete and-work is shared, an and-agent that backtracks into the or-shared part of the tree must always be the team *or-representative*. The worker should thus travel within the team shared or-tree almost as an or-worker would do in a pure or-parallel system.

Since the SBA abides by the *memory independence* principle the other members of the team can continue to execute undisturbed their and-parallel tasks while the *or-representative* executes or-scheduler code, moving along the public part of the And/Or tree looking for or-work to do.

6.6 Generalising the P-WAM towards IAP/ORP

We previously gave an overview of how IAP and ORP should interact in a SBA based combined system. Next, we shall explain how we can adapt the data structures and execution models used to implement binding arrays derived ORP systems such as Aurora, and RAP-WAM [72, 75] derived IAP engines, such as &-Prolog and &ACE

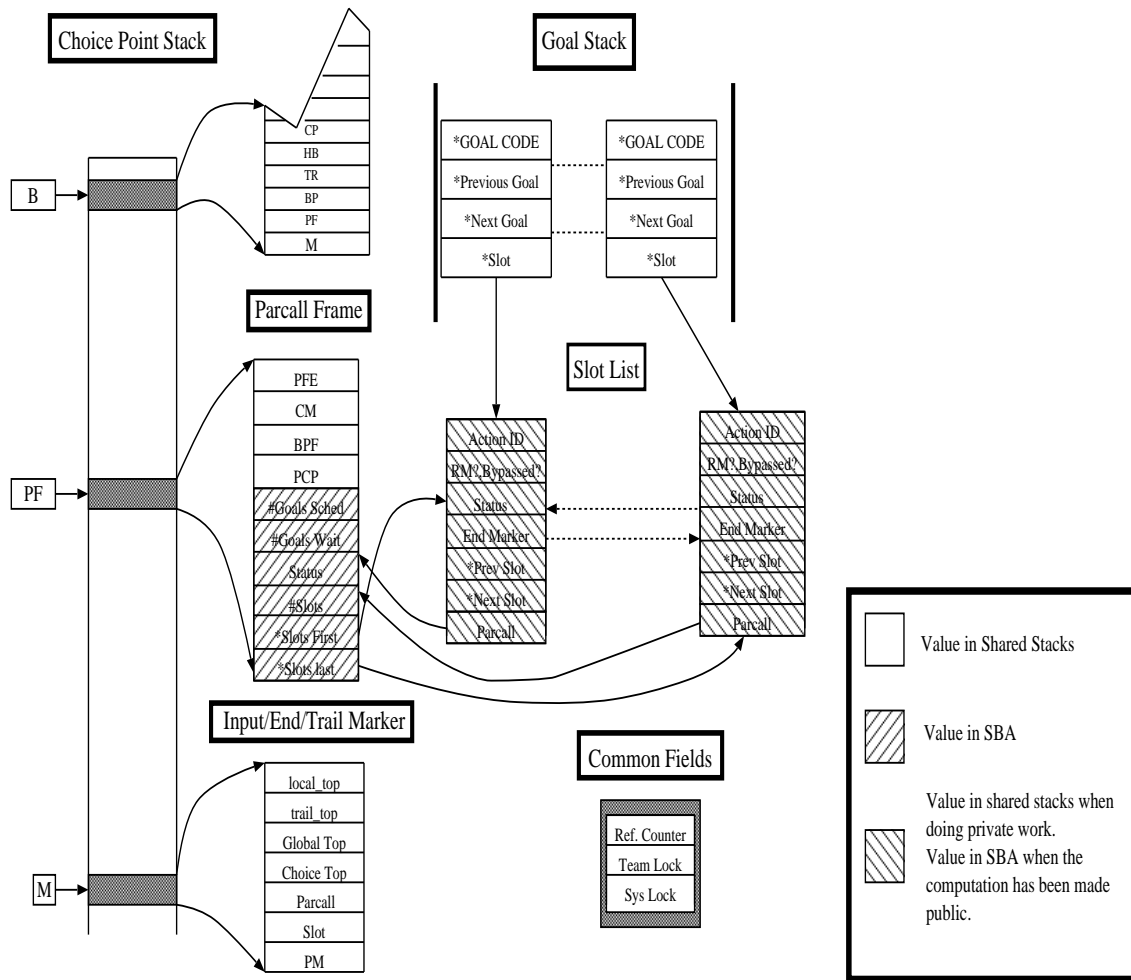


Figure 6.3: Data structures for or- and iap-and Parallelism.

in order to obtain a SBA recomputation based And/Or parallel execution model for Prolog.

Figure 6.3 presents the major data structures involved in the support of And/Or parallelism. In our discussion we concentrate on the issues relevant for the combined system, please consult [25, 72] for the issues in ORP and ANDP systems.

6.6.1 Extra Machine Registers

The abstract machine registers closely follow the WAM registers. We introduce two new registers for IAP. The PF register always points to the current parcall frame should a failure occur. The M register points to the Input Marker of the current iap-computation.

The two registers must be saved in choice-points, but only for shared or-work, as they are known for each and-task. We may always save them in the `try` operations, or we can delay until the team *or-representative* makes the choice-point public. In the latter case it will be up to the or-representative to store the correct values into these fields.

6.6.2 Base Fields

We simplify our design by including a standard set of entries for all structures involved in supporting IAP and ORP, the “*Base Fields*”. These fields provide synchronization and object management. The fields are:

- **Ref. Counter**- describes how many teams are at the moment using the object. For instance, a node in the search tree is or-public if and only if its Ref. Counter is a positive integer, and or-private if zero. Note that if a node is public, all stack segments between this node and its logical ancestor must also be public. The field is needed for recovering shared data structures after backtracking: we can only recover an object and associated stacks when the field drops to 0,
- **Team Lock**- A synchronization lock private to a team. It is needed, for example, to ensure mutual exclusion when taking a goal from a parcall frame. We can avoid unnecessary synchronization by locating the lock in the SBA.
- **System Lock**- global synchronization global lock. It is used, for instance, when updating the reference counter for an object.

6.6.3 Computation slots

Each and-parallel goal is represented by a `computation slot`, that stores state of an and-task. This data structure has the following fields:

- **ActionID**- The Id of the and-agent within the team, if any, that is currently working at or that finished the goal.
- **RM?Bypassed?**- A 2 bit flag field indicating whether or not the computation is rightmost and/or the computation has been bypassed from the computation on its left.

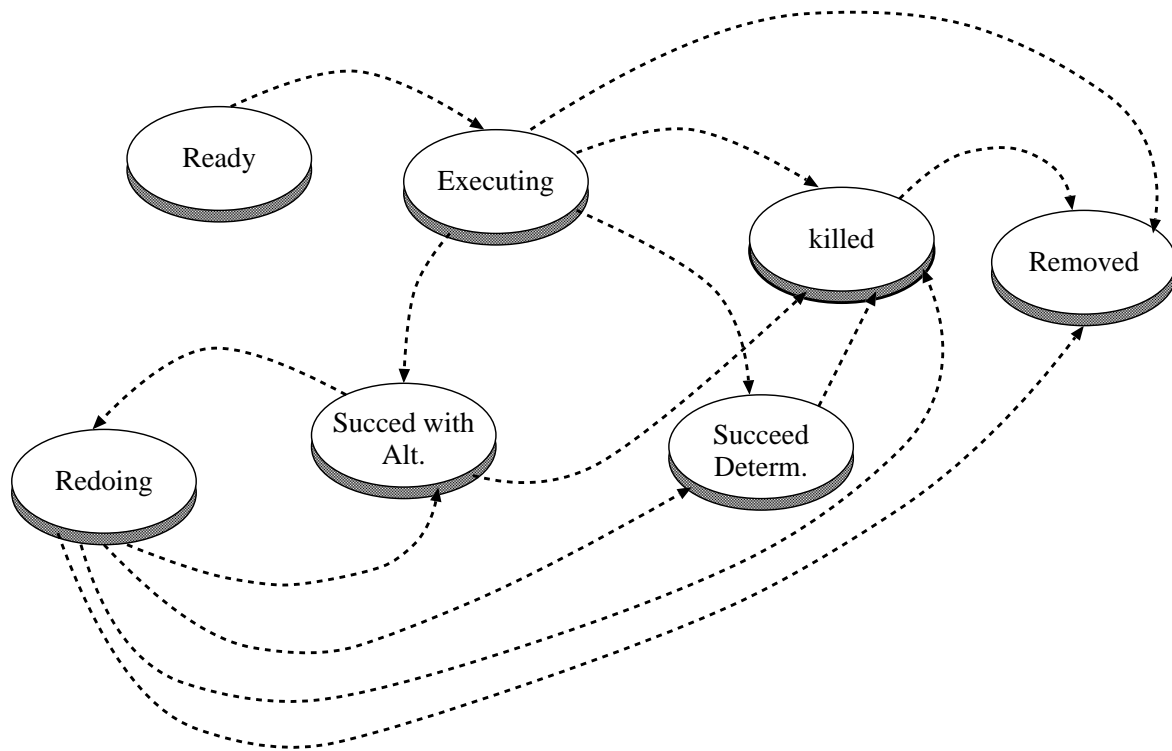


Figure 6.4: Computation Slot states.

- **Status**- current status for the goal. We describe the possible states in some more detail next.
- **End Marker**- If the goal finished or has suspended, a pointer to the end-marker. The End Marker depends on which branch in the search tree we exploit, and thus must be stored in the SBA.
- **PrevSlot**- A pointer to the previous goal-slot, or NULL if this is the leftmost goal.
- **NextSlot**- A pointer to the Next slot in the list. Null if this slot is the rightmost goal.
- **Parcall**- A pointer to the Parcall that “owns” the slot.

Goal States Figure 6.4 shows the life of a goal. Goals start from the **READY** state. This means that there is a goal descriptor in a goal queue for this slot. When an and-worker selects the descriptor, the slot changes its state to **Executing**. Execution may lead to four cases:

1. It enters the **Succeed With Alt** state if the goal succeeds with open alternatives.
2. It enters the **Succeed Deterministically** state if the goal succeeds without open alternatives.
3. The goal has to be killed, and another and-agent changes the state of the slot to the state **Killed**. As soon as the and-agent working in this goal realizes this, it stops the current computation, backtracks, and the state of the slot becomes **Removed**. Tang, Pontelli et. al discuss how REDO and KILL messages are generated and processed when doing backtracking in recomputation based IAP systems [140].
4. if the goal fails, failure is next propagated to the other goals of the parcall. As soon as the goal is rewinded the state of the slot changes to **Removed**.

6.6.4 End Markers

End markers are data structures used to indicate the top of the physical stacks at the end of a goal computation. The **End Marker** field in a goal slot thus points to the end marker of the goal computation. It requires special processing in the presence of ORP. To understand why observe that whenever an alternative is taken from a completed branch by another team the end-marker for this computation, if it succeeds, is going to be necessarily different from the end-marker of the former computation. As a general case when exploring ORP, for each computation slot there could exist more than one end-marker currently active, one in each team where alternatives for the computation succeeded.

To solve this problem an entry for this field is stored in the trail when the goal completes. This way when or-work is taken by another team, that includes this branch, the proper value for the end-marker can thus be stored in the team's SBA location for the computation slot.

6.6.5 Parcall Frames

We create parcall frames whenever the conditions in a CGE [72] succeed and we can start and-parallelism. This structure contains control information for IAP. The following fields describe how the parcall links with the external computation:

- The parcall frame environment PFE points to the local environment when the parallel goals were activated, and it is necessary to continue execution.
- The continuation input marker CM stores the value of the M register when the parcall was created.
- The previous parcall frame pointer, BPF allows propagating failure.
- The continuation of the parcall frame, PCP, is a pointer to the code to execute when the parcall-frame completes.

The PFE, CM, BPF, PCP are established when the parcall started and therefore need not to be copied to the SBA. Other fields control current execution status:

- #Goals_Sched gives the number of goals still to be scheduled.
- #Goals_Wait gives the number of goals that are executed but have not yet reported success.
- Parcall_Status, indicates whether we are in inside or outside mode.
- #slots is the number of goals in the parcall frame.
- slotFirst and slotLast point to the first and last computation slot.

Note that if N_SUC is the number of goals in the parcall that reported success then we have:

$$\#slots = N_SUC + \#Goals_Sched + \#Goals_wait$$

Shared parcalls must store the #Goals_Sched, #Goals_Wait and Parcall_Status in the SBA.

6.6.6 The Goal List

The goal list provides a mechanism for and-agents to share work within a team. In &-Prolog each and-agent has a goal list, which is visible to all other and-agents. Each element of the list is named *goal descriptor* and consists of four fields:

- **Goal Code** points to the code to execute;
- **Slot** points to the corresponding slot;
- **Next Goal** and **Prev Goal** point to a doubly linked list.

Goal lists are private to the team and need not to be stored in the SBA.

The goal list is maintained by two registers, **FG** and **GL**. Both are private to each and-agent. The **FG** register points to private work, and the **GL** register points to shared work, where access must be synchronized. For several optimizations related to the management of this sort of run-queue please consult [41].

Observe that each and-agent in the system has a private memory area from where goal descriptors are first initialized. The area initially consists of free descriptors. Every new and-goal takes a free descriptor from this list, initializes it, and then puts it in the list pointed to by **GL**. Experience with &-Prolog shows that the number of goal descriptors owned by an and-agent can grow dynamically.

6.7 LPCO in And/Or Systems

The &ACE system includes a powerful optimization, last parallel call optimization (LPCO) [140]. The idea is to optimise the common case where we are launching a new set of and-parallel goals such that their continuation returns directly to a previous parcall-frame \mathcal{P} . Often, the computation from \mathcal{P} to the current points was deterministic. In this case, we can include the new goals in the \mathcal{P} instead of creating a new parcall frame. The optimization obtains impressive performance improvements for both space and time [140] and should thus be considered in a combined system.

The &ACE implementation of last parallel call optimization [51] works by adding new computation slots dynamically to a parcall. As such, goals slots must be kept as a doubled linked list, and not as an array as in &-Prolog. LPCO only changes the slot list deterministically. It hence does not require trailing neither the fields affected by LPCO have to be stored in the SBA: LPCO is ORP transparent.

6.8 Summary

We discussed the main implementation issues in integrating IAP and ORP systems. We showed that the orthogonality inherent to using the C-tree model with the SBA data structure allows for a clear separation between IAP and ORP, and simplifies the task for an implementor. As ORP is the least affected, we show how to adapt the major data structures in the RAP-WAM. Major changes are the need to trail status fields, and to store them in the SBA when the computation is or-shared.

Unfortunately, actual experience in implementing this model showed severe difficulties that could lead to an inefficient And/Or system. One first problem was the sheer complexity of the traditional implementation of IAP, namely in the implementation of backtracking. A second issue was the small granularity of the IAP backtracking operations. We were also concerned with the overheads introduced by creating too many parcall-frames, even in the presence of LPCO.

Our experience thus led to the investigation of a different model that could simplify the combined system. Our research was guided by our experience with ORP systems and by our previous analysis of the relevant integration issues. The next two chapters describe this new model, LGSATS, that we believe is much more amenable to integration with a SBA based ORP system.

Chapter 7

Independent And-Parallelism Revisited.

In the previous chapters we have presented execution mechanisms for the combination of the SBA with classical parallel implementation schemes to exploit both IAP and ORP in Prolog. Our proposal for ORP was based on Aurora, whereas our approach to IAP was based on the Hermenegildo's P-WAM [74] and in &ACE [105]. These approaches rely on *parcalls* and *markers* to support IAP.

Our experience has shown two significant issues when using these proposals towards the development of an independent and-parallel system. First, in the original P-WAM design, parcall frames and markers must be built for every parallel conjunction. This incurs costs, both in memory and space. Thus, execution speed may suffer when compared with the execution speed of an optimized sequential Prolog system. Arguably, the high cost for creating and managing new parallel tasks can be reduced through compile time analysis. Indeed, much work has been done in the area of granularity control for and-parallelism by generating program annotations [54, 43] or simply by throttling IAP [129]. Unfortunately, these solutions require sophisticated global analysis tools in order to work well.

The second, and major, issue was a desire to simplify the implementation of IAP, and particularly of backtracking. Although the *Orthogonality Principle* simplifies the integration of IAP/ORP, in practice our system will always be more complex than the sum of its parts. We wanted to take the best advantage of recent experience in the implementation of IAP and ORP systems, towards building a simpler, and hopefully more efficient, IAP/ORP system.

Our work in the design of the SBA made the following key ideas clear: **(i)** we should always take advantage of the analogy between ORP and IAP [103]; **(ii)** we should avoid creating new structures, and instead re-utilize WAM data structures wherever possible; and **(iii)** we should avoid major changes to the compiler, as they will impact system maintainability. We shall assume that the underlying system is WAM based [151], like YAP [42, 112].

We next present what we believe is a novel IAP execution model, fully supporting backtracking and memory reuse, which incurs minimum overhead while executing non-public and-parallel code and designed from the beginning to comply as much as possible with an orthogonal IAP/ORP integration.

Our design starts from a standard WAM based sequential Prolog implementation. We then incorporate the minimal number of changes that we believe are required for an efficient implementation of IAP. Ultimately, we aim at a novel IAP proposal that can be naturally extended to support ORP through the SBA [39] or other approaches, such as the α COWL [111].

7.1 Design Guidelines

Our new model for the exploitation of IAP is called LGSATS, (*Large Grain Sequential And Tasks for SBA*), Our design is based on the following main guidelines:

- And-agents perform as sequential engines for the most part of their execution.
- Make and-work available for parallelism lazily, thus inducing coarse grain and-tasks for parallel execution.
- Avoid creating new specialised structures, such as markers or frames, by re-utilizing WAM data-structures wherever possible. The idea is to simplify system implementation, facilitate IAP/ORP system integration and reduce overheads.
- Avoid major changes to the compiler so that reuse of previous work becomes possible and desirable.

Our key implementation principle is that, in order to make the sequential mode of execution as similar as possible to sequential Prolog, parallel constructs should result in minimum overhead during forward execution. To achieve this goal, whereas traditional

IAP system prepare the data structures required to support parallelism when finding parallel conjunctions, we want to prepare such structures much later, *only when work sharing is needed*. As we shall see, this is possible if the parallel data-structure piggy-back on sequential data-structures. Space for parallel constructs will be allocated as an extension of a corresponding sequential construct during forward IAP execution. However, we will only initialise this space when parallel execution is required.

We therefore propose a lazy strategy for publishing parallel and-work: workers should ignore parallelism for as long as possible. This is an important requirement because access to shared parallel structures requires synchronisation, turning access more costly. We will have to pay a price, as not all possible IAP will be made available within a team. Instead, we focus on guaranteeing just enough to keep the and-agents in the team fully occupied.

These principles take advantages of previous experience in ORP systems, where overheads are reduced by increasing grain-size. To understand how the approach works, visualize a *complete* sequential computation branch (from the root to the leaf of a computation branch of the sequential SLD resolution tree) as consisting of several “pieces”. And-parallelism can be seen as computing pieces in advance and later gluing them together in an orderly way. Classical IAP execution schemes break the tree into as many “pieces” as the tree can be broken into. By making and-parallelism available lazily, LGSATS will “break the tree” into less pieces. This makes average grain-size coarser, reducing the work both in breaking and gluing more pieces together.

Traditional IAP systems benefit from the fact that access to *private* constructs, namely logical variables, does not need to be synchronized. By minimising the number of *public* data structures, LGSATS extends this advantage to conjunctions which would require synchronisation in traditional IAP implementations.

Consider the following example. Most IAP systems place goals that are available for parallel execution into shared data structures, the *run-queues*. It is likely that an and-agent will eventually pick up goals it itself published. Indeed, and-agents often put their goals in the run-queue, only to later pick them up again for execution. This overhead can be minimized if we delay publishing and-work.

7.2 Principles of Forward Execution

We next refine our definition of IAP-execution in order to understand the runtime constructs necessary for an implementation. Remember that in its most basic form, IAP corresponds to taking goals from a conjunction and running them in different computing elements in parallel.

The key concept in our work is *parallel conjunction*:

Definition 4 Parallel Conjunction: a set of sub-goals in the body of a clause that may run in parallel.

We sometimes abuse terminology and use the term *CGE* for parallel conjunction. It should be clear from the context whether we mean a CGE in the original sense or a parallel conjunction. We also define:

Definition 5 Parallel Goal: a goal in a parallel conjunction.

At any moment during execution, a number of parallel conjunctions may be present, and therefore more parallel goals than processors may exist. An allocation function must distribute goals.

Definition 6 And-scheduling: policy of deciding which goals run in which and-agents.

We shall concentrate on the mechanisms that provide goals for parallel execution here, as standard and-scheduling policies should apply.

Remember that a parallel conjunction is a subset of the goals in a clause's body, and that the WAM represents clauses as environments. Our first point is to argue that and-parallel execution should also be represented directly *within* WAM environments:

All information regarding and-parallel execution of a goal is part of the environment of the clause that contains the goal.

Practice has demonstrated[123, 51] that IAP-execution leads to several subtle difficulties. As regards forward execution, two problems must be addressed: memory management and scheduling.

7.2.1 Memory Management

Our key concept will be *sequential-task*, or and-task:

Definition 7 And-task: a maximal set of parallel and/or sequential goals executed in such a way that (i) their computation follows the Prolog sequential order of execution; (ii) they are executed by the same and-agent.

By maximal we mean the and-task is never a proper subset of another and-task. Therefore, a goal always belongs to a single and-task. Moreover, if two goals are sequentially contiguous in Prolog order, or if a rightmost parallel goal and its continuation were executed by the same and-agent the two goals must belong to the same task.

And-tasks are build dynamically. Their number depends on the number of processors and on scheduling. Further, and-tasks may include several parallel goals. Indeed, they may include all goals in a parallel conjunction. If so, the conjunction is run sequentially, and there is no gain in building the data-structures for parallelism. We thus designed our model to minimise such overheads.

Implementation-wise, an and-task begins with the execution of a parallel goal received from the scheduler, or taken from the And/Or tree. Or it may start at a sequential goal, the continuation of a parallel environment. And-tasks end when the and-agent is not able to incorporate the next goal or the environment-continuation into the current and-task, either because another agent executed the goal, or because it had to start a new task for the environment continuation, or because the and-task failed. After closing an and-task, the and-agent must backtrack or look for more and-work within the team.

One first concern when supporting and-tasks is memory management. Remember, each individual and-task is a Prolog computation and its state can therefore be stored in a WAM stack. Unfortunately, simultaneous and-tasks cannot be directly represented within a single sequential WAM stack. To address this problem, most parallel systems give each and-agent a chunk of the total available memory and allow each one to use its chunk to operate as an independent Prolog engine. We shall call this chunk the agent's *stack-set*. While doing forward execution each and-agent will see this local space as a set of WAM stacks that expand and contract in a way very similar to what would happen with a sequential Prolog system.

The history of each and-agent will be the execution of a sequence of and-tasks. Because space for an and-task can only be discarded after we fully backtrack over it, part of

each agent's stack-set will end up being taken by memory segments for *frozen* tasks. System memory is therefore organized as a complex graph of task segments, several per and-agent stack-set. We would like to reduce the number of nodes in the graph by guaranteeing that all goals in the same and-task are represented by a single segment.

The P-WAM [74] works with segments. Two markers, the *start-marker* and the *end-marker* represent the start and end of a segment. Memory management for ORP system such as Aurora, is quite different. An or-task is a set of or-parallel nodes. Memory management for an or-task is performed through a special choice-point, the *sentry-node*. Space is recovered on backtracking only.

We can take advantage of the way memory management is done in an ORP system such as Aurora to propose the use of a special choice-point that will manage space recovery and-parallelism. The *and-task* choice-point node indicates the start and the end of an and-task. This new node is still a *marker* or a *sentry*. LGSATS thus presents two contributions:

- Markers are integrated into standard WAM objects, in this case, choice-points;
- The system is built so that the number of and-tasks should be minimal.

Our reasons for choosing a choice-point node instead of special terms in the global heap or trail are largely pragmatical, namely:

1. IAP/ORP integration – The or-scheduler can more easily be modified to cope with and-parallel constructs within or-public areas. The presence of and-parallelism is indicated by the use of a different choice-point node.
2. The use of the choice-points stack automatically induces trailing and environment protection for older and-parallel constructs.

7.2.2 Control Flow in Forward Execution

Our basic principles and control mechanisms for the runtime exploitation of IAP are:

Default execution Default execution is sequential, as in other multi-sequential systems. In other words, workers try to execute all parallel and sequential goals within the same and-task. On finding a parallel conjunction, the corresponding WAM

environment will have some extra memory cells reserved for the support of parallel execution, should the need arise. To keep pace with our *lazy* approach these fields are initialized only when they are publicized. Therefore, they can be simply *ignored* in the absence of parallelism.

Work Publishing Work Publishing is to be performed *lazily* and on *demand*. After receiving a request for and-work from some other member of the team, workers traverse their environment chain. Private *parallel-environments* with unpublished parallel alternatives are then initialized by setting up the extra fields and adding the yet unexecuted parallel goals to the and-agent's run-queue. If we also take an *hybrid approach* in scheduling, parallel goals can be marked as executable directly on the and-tree. After the completion of this operation, the just publicized goal-slots become visible to the other team and-agents. The parallel environments containing these goals are now *public within the team*.

It is very important at this point to differentiate between two forms of work visibility. Work visible within a team corresponds to and- parallelism and work visible by several teams corresponds to or- parallelism. To the first form of visibility within an IAP/OR system we call *and-public* and to the second *or-public*.

Returning into a public environment This operation will eventually happen if we execute successfully after having published goals. At this point, we will have the following options (remember that we want to maximise and-task size):

1. If the next goal to execute is available, the and-agent should take it and continue to expand its current and-task.
2. If there are other goals available in the parallel environment but the next goal to execute has already been taken by some other and-agent, the current and-task must be terminated. The and-agent should then try to begin a new and-task by picking the leftmost still available goal from this parallel environment. If this fails, the and-agent must call the and-scheduler to look for more and-parallel work.
3. If the and-agent has just executed the rightmost parallel goal and all other goals in the parallel environment have already been completed, then the continuation can be *incorporated* into the current and-task.

4. If the and-agent is reporting success to a goal that is not the rightmost and all other goals in the parallel environment have already been completed, then the and-agent must terminate its current and-task. It then starts a new and-task by proceeding with the execution of the continuation.

Scheduling Issues

Hermenegilo's *no-slowdown* [76] scheduling rule guarantees at most a constant-bound slowdown over sequential execution. One way to enforce this rule is to ensure that all parallel goals "to the left" execute first. To be more precise, we define a partial order for all parallel goals, say $<$, such that $g_i < g_j$ if g_i and g_j belong to the same parallel conjunction and g_i is written to the left of g_j . The no-slowdown rule is guaranteed when g_j can only execute if g_i is actively executing or if it has executed.

The no-slowdown rule is automatically satisfied by the forward execution rules we have already explained. To prove this it is sufficient to show that the leftmost non executed parallel goal will always be taken by some and-task without the need for and-scheduling.

Without loss of generality, call the leftmost still available goal G_i . From our model it is clear that G_i must be > 1 , because the first goal of a parallel conjunctions is always executed in sequential mode (default rule). Moreover, we know that $G_{i-j, j \geq 1}$ have already been taken for execution, otherwise G_i would not be the leftmost still available goal. Suppose now that and agent W_k completes G_{i-j} . If $j = 1$, according to the public environment rule 1, G_i will be executed by W_k without need for and-scheduling. If $j > 1$, according to the Public Environment rule 2, G_i will be picked up for execution by W_k without the need for and-scheduling. We can therefore guarantee the no-slowdown rule.

An LGSATS worker calls and-scheduling at the Public Environment rule 2). One alternative, following ORP systems, is to go up the and-tree. This approach is directly supported by the LGSATS machine instructions we present next when they are executed in a special mode of execution (*and-search* mode (section 7.8)).

The second alternative can be activated when the and-agent reaches the root of the and-tree in and-search mode still without being able to find any and-work. When an and-agent enters this second phase of and-scheduling it starts looking for and-work in the and-agent's run-queues (starting from its own run-queue).

After this second phase of and-scheduling, if the and-agent is still unable to find any and-work, it issues a broadcast request for and-work to the team and goes to sleep waiting for the future availability of and-work. Later, when some other and-agent in the team processes the request and makes some of its private and-work available it wakes up all the and-agents that are sleeping on the availability of and-work. When these sleeping and-agents wake up they can compete for this new and-work by traversing the run-queues.

7.3 Data-Structures for LGSATS Execution

LGSATS agents use WAM stacks. All the data needed to manage IAP parallel execution is placed in run-queues as goal frames, in environments as goal-slots, and in new choice-point nodes. We next discuss each data structure in detail.

7.3.1 Goal-frames and and-agent run-queues

Our model uses goal-frames as key data-structures for and-scheduling. A goal-frame contains a pointer to the code that starts the execution of a parallel goal and a pointer to the parallel environment that owns it.

Each and-agent contains a *work-queue* of goal-frames, divided into a public and private part. When initialized, goal-frames are first pushed into the private part of the and-agent's work-queue. Goal-frames can be taken from the private part with no locking and are transferred to the public part lazily and on demand. Locking should be only necessary when operating on the public part of the run-queue. Each and-agent also maintains a *free-list* of goal frames, each one ready to be initialized and pushed into the private part of the and-agent runqueue. This method of organizing available work is commonly used in a variety of parallel systems with scheduling based on distributed [49] work-queues.

Goal-frames are thus reusable objects that can be transferred from and-agent to and-agent. Work is transferred when an and-agent \mathcal{A} picks work from the public part of another and-agent, \mathcal{B} 's run-queue. In this case the "foreign" goal-frame is moved into the the and-agent \mathcal{A} 's *free-list*. Goal-frames are thus memory objects that can potentially be used by all members of a team.

Due to backtracking operations, goal-frames in the run-queues can be marked *free*,

that is, the code pointer becomes NULL. It is thus necessary to garbage collect goal-frames from time to time. We can implement several policies for the activation of this goal-frames garbage collection procedure. Our first suggestion is to initiate it each time a new goal-frame is necessary and the and-agent's goal-frame free-list is empty.

Observe that since only complete work ends up being or-shared, goal-frames can be allocated from a memory area shared by all and-agents within a team but not necessarily accessible by other teams. We can thus save SBA space by allocating goal-frames from some other memory area private to each team.

7.3.2 Parallel Environments.

In LGSATS we organise the classical “parcall” structure necessary to maintain IAP into two sub-sets. Static data, eg, the number of parallel goals in the parcall, is kept in compiled code, more specifically in the operands of IAP-specialized machine instructions. Dynamic data, that is, the state of execution of the parallel goals, is stored in the clause's environment. We call the extension a *CGE* frame, as illustrated in figure 7.1. Values in CGE frames are only valid for parallel environments.

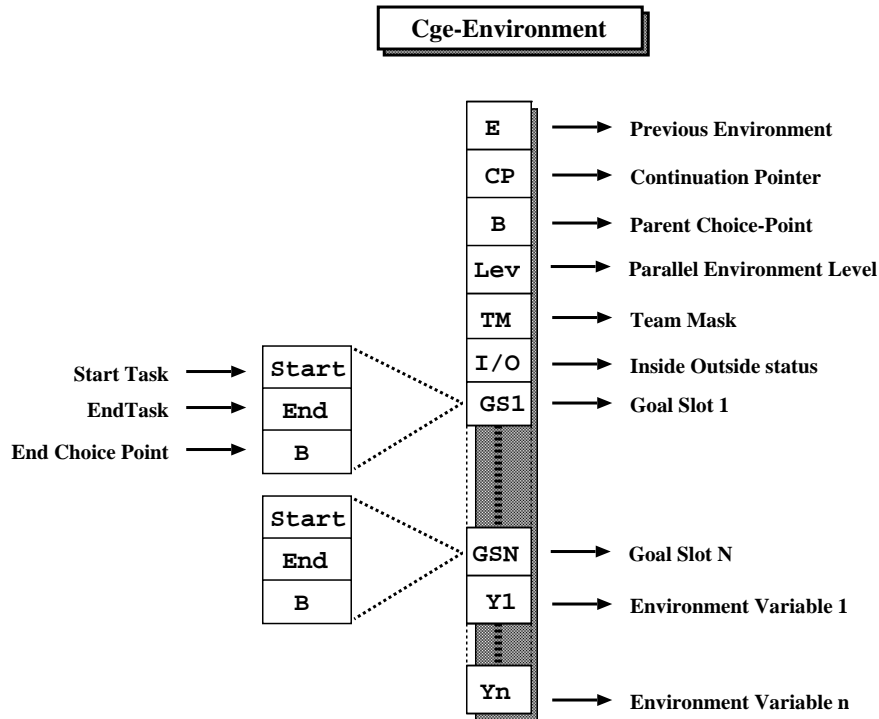


Figure 7.1: A CGE-environment in LGSATS.

Parallel dynamic data maintained in CGE frame are as follows.

Goal-Slots There is one goal-slot for each parallel goal in the CGE, which is used to perform parallel goal management and to integrate parallel goals into and-tasks. As can be seen in figure 7.1 a goal-slot is composed by three memory cells. The first cell (**Start**) is a tagged pointer and represents the execution state of the corresponding goal. The second cell (**End**) is either null (the goal has not been finished yet) or it points to the task-node of the and-task that completed the goal. The third cell is either null or it contains the value held by the B register of the and-agent at the time it completed the goal.

When a parallel environment is and-public the first cell of an and-public goal-slot is a tagged pointer. The tags and respective pointed object types can be one of the following:

READY – The memory cell points to the goal-frame in a run-queue corresponding to this goal invocation. This pointer is used to mark a goal-frame as **INVALID** when **(i)** the goal is picked up from the search tree by the execution of a LGSATS instruction in and-search mode 7.8; or, **(ii)** when, during backtracking, the parallel environment containing the goal-slot is backtracked over.

If goal-frames were not marked as **INVALID** the goal could be picked up a second time from the run-queue. Moreover, this allows for goal-frames to be garbage collected.

TO_BE_LAUNCHED – The memory cell points points to the first WAM instruction that starts the execution of the associated goal. The reason for the existence of this tag is twofold. First, it allows us to include a builtin mechanism in LGSATS to help control the *and-or speculativeness* of and-or tasks (identified in 3.6); secondly it gives us a mechanism for the fast relaunch of and-public goals after backtracking.

RUNNING – This indicates that the associated goal is currently being computed. Further details about the and-task where the goal has been incorporated can be found from the task node pointed by the goal-slot first cell.

FINISHED_ALT – This tag is used to indicate that the associated goal has finished but alternatives in the goal still have to be tried.

FINISHED_DET – This tag is used to indicate that the associated goal has finished with no more alternatives to be tried.

DETERM_PURE – This tag indicates that the goal has finished never creating alternatives nor executing any side effects.

Tasks where all parallel goals have been tagged with **DETERM_PURE** do not need to be redone when we perform outside backtracking. For the same reason, they do not have to be redone even when this part of the tree is or-parallel shared with other teams. The task is a compute once, use everywhere and-task, and should be deallocated only when the CGE-environment where it is contained is fully backtracked over.

Notice that the **READY** and **TO_BE_LAUNCHED** status contain a reference. The **RUNNING** status can be implemented as a pointer to a compound term. The other three status can be simply implemented as integers.

B: Parent Choice-Point When a new parallel environment is allocated the current value of the WAM register **B** is saved into this cell. This value may be required for backtracking in the presence of IAP, as discussed in the next section.

Lev: Parallel Environment Level This field saves the `level` at which the parallel environment is allocated. We need this field because we use *levels* (section 5.4) expanded for IAP (section 7.7) in order to manage backtracking.

TM: Team Mask The Team Mask used for IAP/ORP, it indicates which teams are *or-located* “under” the parallel. It is implemented as a bitmap and it indicates whether or not a parallel environment has been made or-public.

Non-zero team masks indicate that the parallel environment has been made publicly available for ORP and that it can now be seen and used by other teams. Also, every cell in the parallel environment cells should be treated as a *conditional* multivalued variable and therefore all updates (“bindings”) have to be made to their associated SBA position.

I/O: Inside/Outside Status this field indicates the execution status for the parallel environment, that is, whether or not the parallel environment has already been

completed. When a parallel environment is created it is in `INSIDE` status. As soon as all environment goals report success and execution of the CGE continuation starts, the CGE-environment enters the `OUTSIDE` status. This is similar to what happens in the classical IAP schemes.

When backtracking, because we are exploring independent-and parallelism, if we pass through a parallel environment in `INSIDE` status, we must transfer the `fail` operation to the choice-point pointed by this parallel environment's B cell. This will be explained in more detail later.

Conditional CGEs

Notice that if the clause's CGE tests fail, then the environment (if allocated) does not need to incorporate the new extra LGSATS cells. This can be properly accommodated with a compilation scheme that divides the clause code into two branches. One sequential and other parallel. The CGE tests can then be implemented as special instructions that fail by transferring execution to the sequential-branch.

7.3.3 And-Task Nodes.

And-task nodes *represent* and-tasks and differ from a regular WAM choice-point node in two ways:

1. The alternative pointer points to an engine instruction that is called after backtracking to execute memory management code.
2. The A_i registers maintain data required by the LGSATS engine, as we describe next, and not Prolog terms.

Notice that we *do not* place information on actual goals into these nodes, as they exist for, and *only for* backtracking and memory recovery. As is the case in the other choice-point nodes, memory space protected by an and-task node can only be recovered when the node is deallocated.

The main new choice point node is the *TASK node*, or *T-node*. This type of node is created whenever a new and-task is started. The node records both the start and end of the and-task. The and-agent's *CST register* always points to the current T-node.

Task-node allocation is made at task startup. Choice-point fields are initialized with the physical top of all engine stacks as in a WAM `try_me_else` instruction.

The T-node is closed when the sequential task terminates successfully. Task closure data is part of the T-node data structure and contains the physical top of all engine stacks and a pointer to the last WAM choice-point allocated during and-task execution. This last field is `null` if the and-task has been deterministic.

T-nodes are recognised via the alternative pointer, which always points to the instruction *start_of_andtask*. This instruction implements backtracking over and-parallel constructs in a modular way, that is, with minimal interference with the backtracking machinery of the sequential engine that forms the basis of the parallel implementation.

Notice that the T-node already contains a pointer to the CGE-environment in its `E` field. The CGE environment pointed by the `E` field is called the task's *anchor*. The remaining and-task data is kept in the `A` registers:

- **IndexId** [A_0] – contains 2 sub-fields: the lower half stores the ID of the and-agent that started the and-task, where the ID is a small integer; the higher half stores the number of the goal-slot that started the task.

The goal-slot number plus the environment pointer points to slot in the CGE-environment for the goal that started the and-task. We call the indexed goal-slot the task's *origin*. The goal-slot number is -1 when the task is started as the continuation of a parallel environment.

- **PhysAncest** [A_1] – Is a pointer to the previously physical allocated and-task memory segment. Used for memory recovery attempts during backtracking. We will see later that and-task nodes also act as memory segment headers for memory management.
- **RefCount** [A_2] – Number of other and-tasks nodes whose environment anchor physically belongs to this and-task. This is needed because we can only reuse the space used during the and-task computation after backtracking if this value reaches 0 (section 8.12).
- **StateExec** [A_3] – indicates the current state of the and-task. It points to a LINK node if the and-task is being continued after outside backtracking or suspension. It is initialised to `NULL`.

Similarly to what happens with pointers in goal-slots, we tag **StateExec** to indicate the task state. During forward execution the valid tags are `RUNNING`,

SUSPENDED and COMPLETED . During backtracking they are TO_BE_KILLED, BEING_KILLED and KILLED.

- **MinRL** [A_4] – maintains the minimum reached level (section 7.7.2) during the and-task computation. This value indicates how high in the and/or- tree a task has been able to climb. We will see in the next chapter that it plays a crucial role in our new backtracking scheme.
- **MaxRL** [A_5] – Is used to maintain the maximum level reached during the and-task computation. Used to correctly maintain in an efficient way iap-levels (section 7.7) for the continuation of CGEs
- **CP_End** [A_5] – set to be the physical top of the and-agent’s choice-point stack at and-task closure. This value is needed during backtracking to determine whether or not a choice-point B is located within an and-task.
- **Tr_End** [A_6] – set to be the physical top of the Trail at task closure. This value is needed during backtracking to determine the end-point of the trail-segment that needs to be unwound when the and-task is fully or partly undone.

The **CP_End** and **Tr_End** fields were already present in the *close-marker* data structure of the IAP classical schemes [74].

The *LINK node*, or *L-node*, is a small variation of the T-node, created when a suspended task is resumed or when an and-agent backtracks into an and-task not located at the top of its stacks. In other words, a L-node indicates the *resume* of an already existing and-task while a T-node indicates the beginning of a completely new and-task.

A L-node serves as a bridge between old memory areas and the new memory area that is going to be used after resuming work for the and-task. It contains the current and-agent stacks physical tops, a pointer to the previous LGSATS node, a pointer to the and-agent’s previously physical allocated and-task memory segment and a pointer, initially NULL, to the next L-node. The last field is necessary because there can exist more than one L-node gluing together non-adjacent stack segments belonging to the same and-task.

LGSATS also extends regular choice-point to contain an extra field, the *minimum reached level*. The use and meaning of this field is later explained in Section 7.7.2.

7.4 New Machine Registers

We have shown that two entities play a major role in our execution model: *and-tasks* and *goals*. The latter are present in all major classical schemes for IAP execution of Prolog but and-tasks are a new facet of LGSATS as they avoid goal management and should increase average grain-size.

LGSATS has some new machine registers, the *CST*, to keep track of the current sequential task, the *ASE* (And-Parallel Sentry Node), and the *PASH* (Physically Allocated Segment Header). We describe them in detail next.

CST: Current Sequential Task Register The CST register is a pointer to the current and-task's T-node.

ASE: And-Parallel Sentry Environment The *and-parallel sentinel environment* (ASE) points to the *youngest* and-public environment within the current and-task. It is mainly used to:

- Determine whether the current and-task computation is within the and-public or and-private part of the and-tree.
- Publicize and-work. When an and-agent intends to publicize and-work it just has to follow its current environment list until it reaches the ASE. All parallel environments below the ASE are candidates for and-publication. It is up to the scheduling policy to determine how many are going to be made and-public.

PASH: Physically Allocated Segment Header Each and-task is composed by a set of linked memory segments of consecutive memory cells, one per WAM stack. This memory organization is explained in more detail in section 7.10. It addresses the well known *trapped goal problem* [73], whilst implementing suspension and resumption of and-tasks.

In order to support adequate memory management, each and-agent maintains an extra *internal* register, the *PASH* register, that indicate the physically top allocated *segment header*. Each time an and-agent needs to allocate a new memory segment it checks to see if the *PASH* memory segment is still in use. If this is not the case then the and-agent follows the physical linked list of memory segments that start at PASH until

a memory segment that is still in use is found. A new memory segment can then be immediately allocated after this last busy segment and all other and-agent WAM registers that refer to stack physical tops updated accordingly. The *PASH* register is next made to point to this *opened* memory segment

Level The *current iap-level* indicates the current and-agent's position within the and/or- tree. Notice that this register is similar to the *level-top* register that was used in our SBA based or- system (section 5.4).

Contrary to what happen with other WAM registers, LGSATS registers do not need to be saved into each new WAM choice-point. The *CST* and *ASE* registers do not need to be saved in choice-points because their values can be dynamically determined from the and-or tree at the end of each backtracking operation. The *PASH* register is internal to each and-agent, so it does not make sense to save it in a choice-point.

7.5 Compiling forward execution.

We next introduce the new WAM-like abstract machine instructions we need to support LGSATS, namely: *allocate_pcall*, *start_cge*, *check_par* and *close_pcall*. These instructions assume that the compiler allocates extra cells for parallel environments.

The basic compilation algorithm is as follows:

1. The leftmost CGE goal is compiled as if a normal sequential goal;
2. Each parallel goal has a code block that starts with a *check_par* instruction, followed by the standard WAM code that prepares the arguments for the goal call and closing with a *call* for the goal execution.
3. parallel conjunctions are closed with a *close_pcall* instruction.

Each parallel goal code block is disposed in a way such that after executing *call* the CP register points to the *check_par* instruction corresponding to the parallel goal immediately to its right or to the *close_pcall* instruction that *closes* the CGE. This code disposition allows us to efficiently recognize at runtime the parallel environments within the current active WAM environment chain:

Definition 8 An environment $EP = E(E)$ is parallel if $E(CP)$ points to a `check_par` or to a `close_pcall` instruction.

It follows from our compilation scheme and from the WAM's execution flow that when we end up executing a `check_par` or a `close_pcall` these instructions are at the start of the continuation of the previous adjacent parallel code block. Note that this adjacent block corresponds to the parallel goal g_i immediately to the left of this parallel instruction. We can thus successfully close the goal adjacent to the left g_i each time a `check_par` or a `close_pcall` is executed.

The organisation of our code also implies that all and-agents must pass through the `close_pcall` in order to execute the continuation of the CGE. This instruction thus acts as a parallel *join* where all and-agents must synchronize until all parallel goals are terminated. CGE continuation starts when the and-agent that reports success to the last parallel goal in the CGE executes `close_pcall`.

LGSATS instructions in a CGE also form a linked list through the second argument. This list is used when searching for new and-work within the CGE. A `check_par` always tries to grab and-work, and if the current slot is already busy it continues to search for and-work in the next cge slots until a `close_pcall` is found.

Next, we introduce these instructions in some detail through examples.

7.5.1 CGE with no sequential goals to the right.

Consider first the case where the CGE has no goals to the right.

$$a(X) \leftarrow (b(X) \& c(X) \& d(X)).$$

This clause is compiled as shown in Figure 7.2. To better understand this code we next explain in detail what each new instructions does. In what follows the word *CGE* is going to be used interchangeably with *parallel-environment*:

allocate_pcall *n_goal_slots*, *n_environment_variables* This instruction is similar to a WAM `allocate` instruction. It allocates space for an environment with:

$$5 + 3 \times \text{n_goal_slots} + \text{n_environment_variables} \text{ CELLS}$$


```

        allocate_pcall 3,1
        get_var Y15,A1
        call b/1

        check_par 2, Next_par, 3
        put_val Y15,A1
        call c/1
Next_par:
        check_par 3, PCP, 3
        put_val Y15,A1
        call d/1
PCP:
        close_pcall 3
        deallocate
        proceed

```

Figure 7.2: Code for CGE With No Sequential Continuation

Observe that at the moment of its creation this parallel environment is not yet visible by the other team members and environment space is not initialised. In the presence of an environment allocated by this instruction the WAM compiler must be modified to add $5 + 3 \times \text{n_goal_slot}$ cells to each Y_i register index.

Whenever a new parallel environment is allocated by this instruction, the *and-agent Level* register is incremented and the new value saved in the parallel environment $E(\text{Lev})$ cell.

check_par *goal_slot, next, n_slots* This instruction is the key to forward execution in our model. It can execute in two different *modes*, the *and-continuation* mode, and the *and-search* scheduling mode.

The following definition connects a parallel environment slot with a worker executing a **check_par** instruction:

Definition 9 If execution is within the code block of a parallel goal G , the Instruction Goal Slot, or IGS, is G 's parallel environment's goal-slot.

The next two definitions allow to us to separate public and private environments:

Definition 10 A parallel environment E is *and-public* iff:

$$E \notin \text{CurrentTask} \vee E \leq \text{ASE}$$

Definition 11 A parallel environment E is *and-private* if it is not *and-public*.

The `check_par` instruction is executed in *and-continuation mode* iff the parallel goal code is reached from a continuation within a normal WAM flow of execution. If the parallel environment is also *and-private*, the IGS goal is immediately *incorporated* (bypassed) into the current and-task without the need for extra synchronization

However, if the parallel environment is already *and-public*, the and-agent must check the value held in the IGS:

- If it is a pointer to a goal-frame in a run-queue the and-agent should try to “acquire” it. If it succeeds, the goal is *incorporated* into the current task. To do so:
 1. The goal must be locked and removed from the parallel run-queue;
 2. The state of the goal moves to `RUNNING`;
 3. The and-agent tests whether the level for this parallel environment is smaller than the minimum level for the current and-task. If this is the case then the minimum task level value must be updated in the current and-task’s T-node and at B.
- If the slot is marked as `TO_BE_LAUNCHED` the and-agent atomically tries to bypass the goal into the current and-task. No goal-frames are involved as the goal is directly taken from the and-tree.
- If an and-agent executing `check_par` in standard mode is unable to bypass the IGS then it must close its current and-task and enter and-scheduling by executing the instruction labeled by `next` in *and-search* mode. The compiler will make sure that the instruction labeled `next` is either a `check_par` or a `close_pcall` instruction.
- If it is already marked as `RUNNING` or otherwise terminated the and-agent closes its current and-task and executes the instruction labeled `next` in *and-search* mode.

When an and-agent executes a **check_par** instruction in *and-search mode* (section 7.8) it first checks if other goals in the conjunction are available. This is implemented by moving to **next** and testing whether **next**'s IGS has a goal to be grabbed. If this is the case it tries to grab it and start a new and-task. If it succeeds it changes the execution mode to *and-continuation* and starts a new and-task with the execution of the next instruction. This new task *origin* is the IGS goal and its task *anchor* is the current parallel environment.

If it fails to grab the IGS then execution continues in *and-search* mode by moving to the instruction pointed to by the label **next**.

close_pcall n_goal_slots – This instruction implements parallel closure of the parallel environment. It is reached as the continuation of the rightmost goal or by an and-agent in *and-search* mode.

- If there are no more *live* goals within the CGE:
 1. **close_pcall** executed in *and-continuation* mode – This can only happen if the and-agent has just finished the rightmost goal. The parallel environment is closed and the continuation bypassed into the current and-task.
 2. **close_pcall** executed in *and-search* mode – The parallel environment is *closed* and its continuation executed within a newly allocated and-task.
- If there are still unfinished goals within the CGE then the and-agent follows the environments linked list (it follows the E(E) pointers) until E(CP) points to a parallel goal (it must be a **check_par** or **close_pcall**). The and-agent then enters or continues and-scheduling by “jumping” to this instruction in and-search mode.

To actually *close* a parallel environment the and-agent must perform the following actions:

1. The parallel environment I/O's state is updated to **OUTSIDE**.
2. The parallel environment's **max_reached_level** must be determined and transferred to the continuation. This value is obtained as the maximum of the **MaxRL** value in the T-nodes of the and-tasks that terminated each parallel goal.

The and-agent then sets its `Level` register with this value and proceeds with the continuation.

3. The deepest rightmost choice point of the parallel environment (DRM section 7.7.2) is determined and passed into the continuation as the new value of the and-agent `B` register.

After completion there is no special treatment for parallel environments. Complete parallel environments are regular environments during forward execution and may be deallocated if not protected by a WAM choice-point. This means that if a parallel conjunction executed deterministically in the same and-task, space for the parallel environment may be reused during forward execution. Thus in LGSATS WAM LCO (Last Call Optimization) also applies in parallel environments.

7.5.2 CGE with Sequential Goals In the Continuation.

The second case for CGE compilation contemplates the case where the CGE has a continuation within the same clause. This is implemented in the same way as in the WAM: we perform environment trimming over the parallel environment by “logically” deallocating it before the execution of the rightmost sequential goal.

Consider the following clause:

$$a(X) \leftarrow (b(X) \& c(X) \& d(Y)), e(Y), f(Y).$$

This clause is compiled into the code shown in figure7.3.

In this example we have two sequential goals (`e(Y)`, `f(Y)`) in the continuation of the CGE. The sequential goals need to access variables from parallel goals. Otherwise the compilation is as usual.

7.5.3 CGE with Sequential Goals to the Left.

The last case is when we have goals to the left of the CGE, as in:

$$a(X) \leftarrow b(X), (c(X) \& d(X) \& e(X)).$$

```

        allocate_pcall 3,2
        get_var Y15,A1
        call b/1
        check_par 2, next_par, 3
        put_val Y15,A1
        call c/1

        check_par 3, PCP, 3
        put_var Y16,A1
        call d/1
PCP:
        close_pcall 3
        put_val Y16,A1
        call e/1
        put_val Y16,A1
        deallocate
        execute f/1

```

Figure 7.3: Code for CGE With Sequential Goals In the Continuation

In this case we need to introduce the last and at the same time the simplest of the LGSATS instructions: the `start_cge` instruction copies the current value of the B register to the parallel environment's B field.

Let us analyse why we need this instruction. Suppose that goal $b(X)$ is non-deterministic, as illustrated in figure 7.4. Goal $b(X)$ thus creates at least one choice-point, say the last choice-point is B_2 . Goal b 's *iap-level* is greater than the *cge-environment's* *iap-level* because the environment has been allocated first. The parallel environment B field will be initialised to point at choice-point B_1 .

Remember that in any IAP Prolog system, execution of the leftmost CGE goal in a clause can only start after the execution of all sequential goals to its left. This must be the case because execution of the leftmost parallel goal, c , may depend on the execution of these goals. Therefore we ignore the parallel environment until we finish with goal $b(X)$. Consider now we entered the parallel conjunction, as shown in the right-hand side of figure 7.4. When and-agent W_2 starts a new and-task with goal d it sets its B register by copying the value of the B value currently held at the parallel environment. Without the `start_cge` instruction this value would have remained pointing to B_1 . Thus, should this new-and task later fail, the goal would backtrack to B_1 instead of B_2 : we would miss the alternatives available in B_2 .

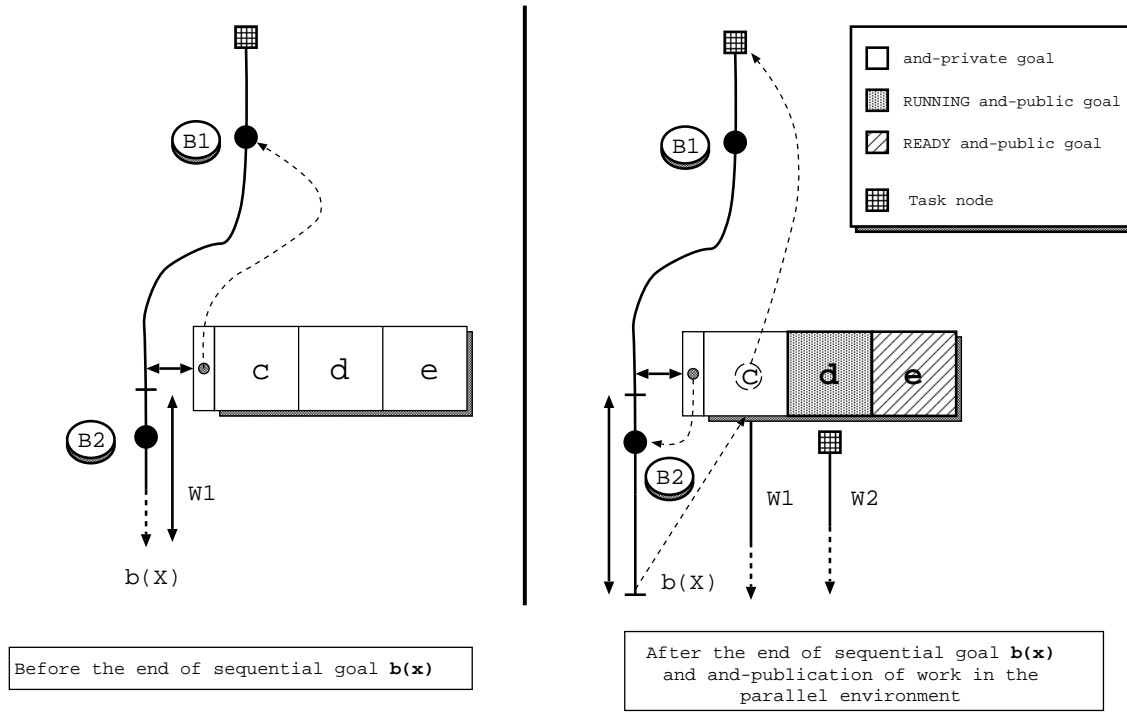


Figure 7.4: Illustrating the need for the `start_cge` LGSATS instruction.

The `start-cge` instruction addresses this problem by separating the environment sequential and parallel scopes. Intuitively, it corresponds to splitting the body of the clause into sequential and parallel components. The compiled code is shown in figure 7.5.

As always, the first `check_par` instruction corresponds to the second parallel goal, not the first: in our model parallel goal c always starts immediately after the termination of the sequential goal immediately to its left (goal b). The first parallel goal therefore does not need a `check_par` instruction. Our design provides the desirable side-effect of hiding parallelism until execution of goal c starts: the environment $E(E)$ is parallel only when $E(CP)$ is a LGSATS instruction, and this can only happen after we started execution of c . We thus guarantee that the parallel goals in environment cannot yet be parallelised, exactly what we need for this case.

Our implementation holds when b is non-deterministic, because if we ever need to backtrack into a choice-point from b , say B_2 , we will update the continuations in such a way that the parallel environment is once again hidden. It will only be reestablished by the next call to the parallel goal c after we computed b 's next alternative.

To finalize observe that since the execution of the sequential goal b can lead to the

```

        allocate_pcall 3,1,par1
        get_var Y15,A1
        call b/1

        start_cge
        put_val Y15,A1
        call c/1

        check_par 1, par3, 3
        put_val Y15,A1
        call d/1

        check_par 2, PCP, 3
        put_val Y15,A1
        call e/1
PCP:
        close_pcall 3
        deallocate
        proceed

```

Figure 7.5: Code for CGE With Sequential Goals to the Left.

execution of other CGEs and therefore to the creation of other and-tasks, there is no guarantee that the and-task that creates the parallel environment where the goal-slot for goal *c* resides is the same and-task that terminates *c* execution. This is the main reason why the leftmost parallel goal in a CGE still needs a goal-frame in the parallel environment.

7.6 LGSATS Overheads

We next argue that *and-private* execution of parallel environments in LGSATS has *very little* overhead when compared with a purely sequential execution. Our argument proceeds by looking at each LGSATS instruction in detail:

allocate_pcall The main differences over **allocate** are (i) the extra space that ends up being reserved for a parallel execution; (ii) the maintenance cost for the **Level** register and **MaxRL** and-task node field. The **Level** register is incremented and then saved in the newly allocated parallel environment. The **MaxRL** and-task node field is just incremented. This should result in very little runtime overhead.

Remember that most of the parallel environment is only initialized when and-work is publicized. This *lazy* initialization of goal-slots is facilitated by the existence of the *and-parallel sentinel environment*, or **ASE**, register that points to the oldest public environment within the current and-task.

check_par This instruction is not present in the sequential code. We will consider overhead incurred during *and-private* forward execution:

1. Perform the test

$$E \in CurrentTask \wedge E \geq ASE$$

that succeeds when the current cge-environment is still and-private.

2. Verify the *minimum reached level* in B's choice-point and update the value if necessary. This operation does not need locking if $iap_level(B) > iap_level(ASE)$, as it would be the case in a purely sequential execution. It needs locking otherwise, because under those circumstances several and-agents may be referencing this choice-point at the same time.

Notice that while the and-agent is positioned within and-private cge-environments it does not have to update the and-task node *minimum reached level*. This happens because all and-private segments belong to the current and-task and iap-levels within an and-task increase monotonically each time a choice-point or cge-environment is allocated. Since the and-task node predates all these data-structures its *minimum reached level* is guaranteed to be smaller while execution remains within the and-private part of the and/or- tree.

These operations can obviously be done in constant time with very little overhead.

close_pcall If the current parallel environment is still *and-private* then the continuation of the CGE is automatically bypassed into the current task by continuing forward execution with the following instruction. Besides the *and-private* test, the only extra work that this instruction ends up performing is to update the backtracking I/O status of the parallel environment to **OUTSIDE**.

Because all parallel goals were bypassed into the same and-task we do not need to determine the environment **DRM** choice-point nor the **MaxRL** for the continuation, as happens in the more general case.

Conclusion The new instructions impose very little overhead during and-private execution. This, together with the fact that on average the task grain size should be substantially larger than in the classical IAP execution models provide two of the three more important contributions of LGSATS. The third is the new backtracking scheme for IAP presented in the next chapter.

As a final remark observe that the use of LGSATS instructions still allow the application of *environment trimming* during compilation because the location of goal-slots cells in the environment is made to precede the location of permanent variables.

7.7 And/Or Tree IAP-Levels

Section 5.4 showed how we can use the notion of choice-point *level* in the or-tree to reliably implement variable-age. LGSATS takes the notion of levels a step further. We expand on the notion of or-levels to iap-levels by incorporating not only choice-points but also parallel environments in *level* tallying. Iap-levels generalise or-levels, and thus can still be used to obtain variable age in an and/or- SBA based LGSATS system.

The notion of iap-level is also fundamental for the dynamics of the novel backtracking mechanism that we present in the next chapter because it allows us to quickly determine the set of parallel goals to be relaunched after failure, or whether we need to continue to propagate failure.

During forward execution each and-agent is located at a certain iap-level: we assume that every choice-point node and parallel environment has a well defined iap-level found as the agent's iap-level at allocation time.

The iap-level position of an and-agent is given by the following rules:

1. The iap-level of computation tree's root is 0. The initial and-agent thus begins at iap-level 0.
2. Whenever allocating a new parallel-environment or choice-point, the and-agent's current iap-level is incremented by one. The incremented value becomes the iap-level of the new parallel environment or choice-point node.
3. New and-tasks always start at the iap-level of the parallel-environment from where we took the task starting goal.

4. If the `check_par` instruction bypasses the corresponding goal into the current task, then the and-task's current iap-level is set to the iap-level of the bypassed goal parallel environment.
5. When an and-agent enters the continuation of a parallel environment its iap-level will be one plus the maximum iap-level reached during the execution of all and-tasks that fall within the scope of the parallel environment.

Remember that there is a basic general completion property implicit to IAP systems: the continuation of a parallel environment can only proceed when the all parallel goals have completed. This property guides the compilation and execution of LGSATS instructions. In LGSATS a goal is closed by the execution of the `check_par` corresponding to the goal immediately to its right, or during the execution of the `close_pcall` if the goal is rightmost. The goal that is terminated when this instruction is executed is named the *instruction closing goal*.

The completion property guarantees that when we close a goal, all computation activities within its execution scope are terminated. This, and the previous rules, justify the following property:

Definition 12 Iap-Level Completion Property – Whenever an and-agent moves up to $iap_level(i)$ all descendant cge-environments of the closing goal with $iap_level(j) > iap_level(i)$ must be completed, that is, in `OUTSIDE` status.

We shall rely on this property to implement parallel backtracking.

7.7.1 Moving In The And/Or-Tree

We next use iap-levels to precisely define what we mean when we say that an and-agent moves up or down in the and/or-tree. We use the notation that $iap_level(agent_j, I_i)$ is the iap_level at which $agent_j$ executes instruction I_i . Also, suppose that the and-agent is executing the instruction stream $S = I_1, I_2, \dots, I_k, I_{k+1}, \dots, I_n$.

Definition 13 An and-agent $agent_j$ *travels down* the and/or-tree at instruction I_k iff $iap_level(agent_j, I_k) < iap_level(agent_j, I_{k+1})$.

An and-agent *travels up* the and/or tree at instruction I_k if $iap_level(agent_j, I_k) > iap_level(agent_j, I_{k+1})$

Notice that if an and-agent fails to bypass new and-work, it will enter into the scheduler and move up and down the and/or-tree in order to look for and-work. We do not consider scheduler tree movements in the current discussion.

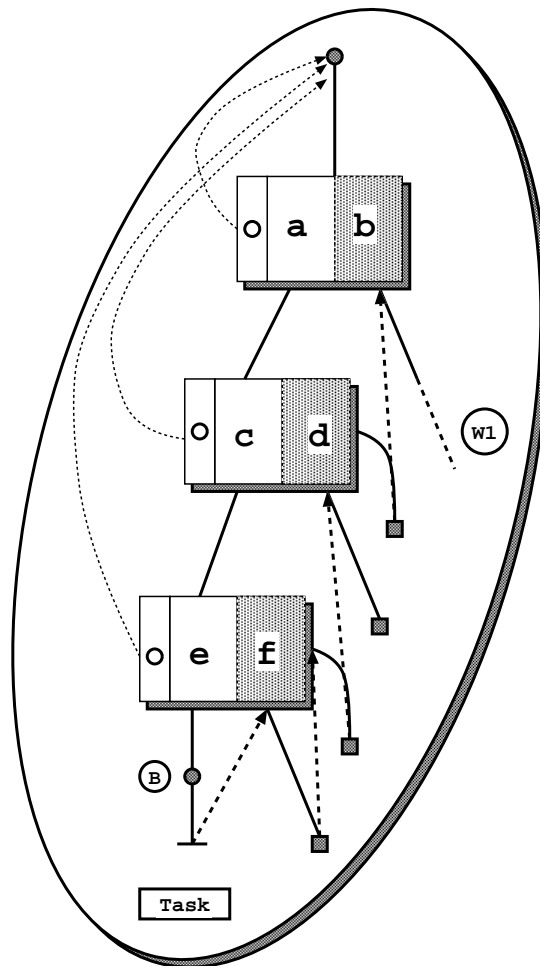


Figure 7.6: Moving up the and/or tree within the same sequential task.

Figure 7.6 shows that worker w_1 has been able to move up the and/or- tree by bypassing the continuation of parallel environments $(e|f)$ and $(c|d)$ and bypassing goals f , d and b .

7.7.2 Maintaining the *Minimum Reached Level*

The *minimum reached level*, or *MRL*, indicates how high in the and/or- tree a task has been able to climb. This value is maintained in the T-node's *MinRL* field and at B's *MRL* field.

This value may be updated when:

- The and-agent *moves up* within the and/or-tree.
- Each time the continuation of a parallel-environment is bypassed into an and-task, the parallel environment's iap-level is compared with the *MRL*. If the environment's iap-level is smaller, the *MRL* is set and saved to B and to the T-node.

When the sequential-task terminates the T-node's *MinRL* indicates the highest point in the and/or-tree reached during its execution.

There is a danger that several agents may try to update the *MRL* field of B at the same time. This may happen if several independent deterministic tasks in a continuation point to the same B, as this field indicates the minimum iap-level reached by all the tasks that will backtrack to B. This situation is illustrated in figure 7.7: all parallel environments except |A| and |B| backtrack to choice-point B_2 .

When an and-agent starts to execute the continuation of a parallel-environment it can easily determine the deepest-rightmost choice-point ($\text{DRM}(\text{PE})$) that falls within the computation scope of the parallel environment. In figure 7.7 this is choice-point B_2 . The $\text{DRM}(\text{PE})$ can be determined efficiently because each time a parallel goal completes it saves its B register in the goal-slot. The *completion-property* guarantees that in the completed parallel-environment PE, the $\text{DRM}(\text{PE})$ can be found by traversing the completed parallel environment goal-slots from the right to left, and stopping at the first slot where the saved choice-point's level is greater than PE's. If execution was deterministic such a choice-point does not exist and the and-agent that starts executing the continuation will set its B register to $\text{PE}(\text{B})$ and the $\text{DRM}(\text{PE})$ minimum reached level (*MRL*) to $\text{iap_level}(\text{PE})$.

to OUTSIDE.

If an and-agent in *and-search* mode manages to update the I/O of the parallel environment:

- We know that the finished goal was not rightmost. Otherwise we would be in *and-continuation* mode. Notice that even in *and-continuation* mode of execution the and-agent still has to grab a lock for the I/O field before gaining the right to the continuation.
- A new and-task must be allocated for the continuation. The new and-task node goal-slot *origin* is set to -1 to indicate that this task sits in the continuation of a parallel environment. This allows us to preserve the implicit Prolog linkage order among completed and-tasks and continuations.

7.9 Publishing And-Work.

And-agents executing and-tasks may each possess a certain amount of *private* and-work. This work is located at the still and-private cge-environments located in the environment list that starts at the environment pointed out by E and ends (but does not include) at the parallel-environment pointed out by the ASE register.

When an and-agent is not able to find and-work in its own stacks nor at the team run-queues, it issues a broadcast message to the team asking for and-work to do. All active and-agents that receive this message estimate the current amount of *private* and-work they currently possess. It is then up to the scheduling policy to determine how many of the candidates private parallel environments, between E and ASE are going to be made and-public.

Remember that an environment PE is parallel iff there exists an environment E such that $E(E) \rightsquigarrow PE \wedge E(CP) \rightsquigarrow (\text{check_par})$

Before starting to publish and-work, the and-agent must decide which one of the and-private and-parallel environments is going to be the new and-parallel sentry environment (N_ASE) and then and-publish all and-work available from this environment until the current and-sentry is reached.

Observe that all parallel goals available for publication have `check_par` or `close_pcall` instructions as the current continuation, including the one that is pointed out by a

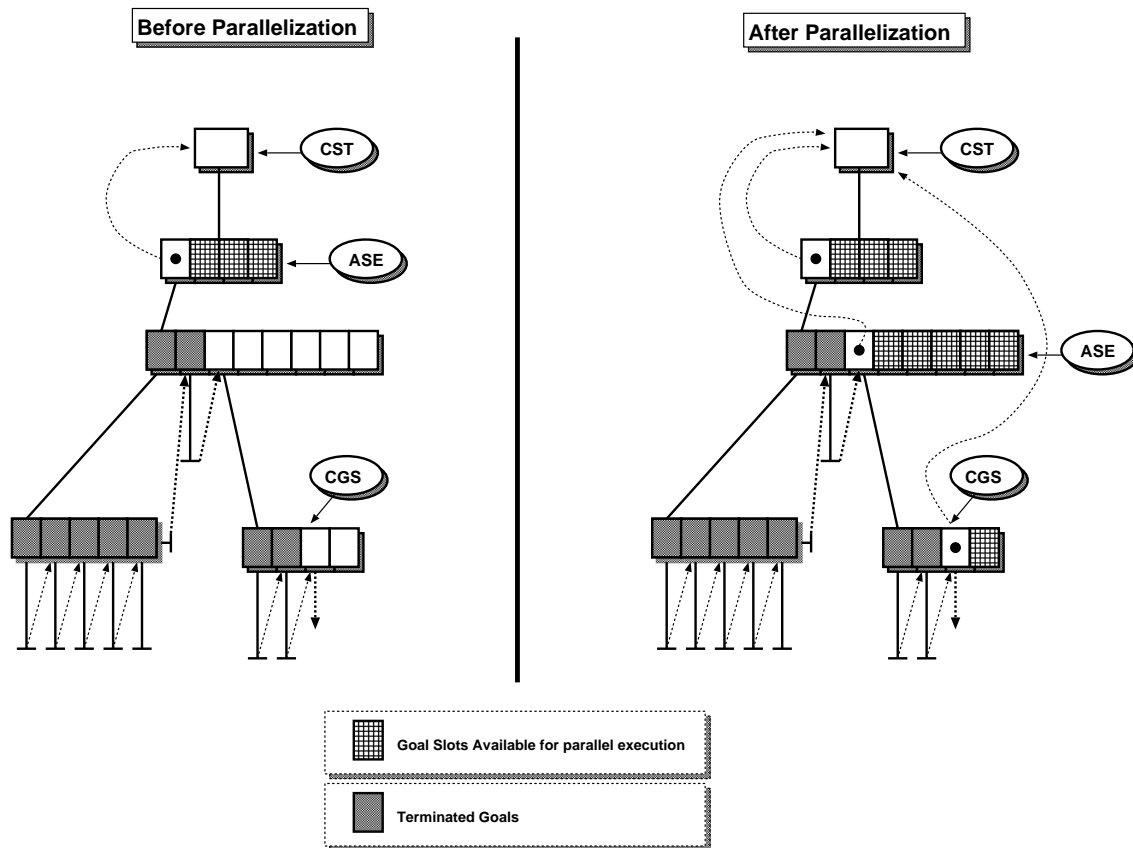


Figure 7.8: Parallelizing cge-environments in LGSATS.

$E(CP)$ (as in the definition). We can thus use a technique analogous to the LGSATS and-scheduling mode provided by the *and-search* mode of execution by defining a new LGSATS execution mode, *and-publicize* that uses P' as a program counter and E' as current environment. The algorithm then proceeds as follows:

1. First, set E' to N_ASE , and P' to the $E(CP)$ LGSATS instruction that identified N_ASE as a parallel environment. Initialise the array GF of goal-frames to be empty and set execution mode to *and-publicize*.
2. While $E' \neq ASE$:
 - Execute a `check_par goal_slot,next` instruction by allocating a new goal-frame pointing to the next WAM instruction in the program code and marking the new `goal_slot` as `READY` with a pointer to the allocated goal-frame. Push the goal-frame into $GF[]$ and proceed in *and-publicize* mode to the instruction labeled in `next`. Remember that a compiler always makes certain that this is a `check_par` or `close_pcall`.

- Execute all other instructions by setting $E' = ENV(E)$ and $P' = CP(E)$.

The algorithm thus stops when reaching the and-sentry ASE . At this point two final steps remain:

1. Consider a list $g_1, \dots, g_i, \dots, g_n$ of goal-slots. Goal g_i belongs to this list *iff* its goal-slot belongs to a parallel environment PE located between N_ASE and ASE and is positioned immediately to the left of the IGS of the *continuation* that marks PE as a currently active parallel environment. This list corresponds to the list of *non-terminated* goal-slots that belong to the current and-task but have yet to be made and-public. This is illustrated in figure 7.8.

We make these goal-slots *and-public* by marking g_1, \dots, g_n as a **RUNNING** tagged pointer to the current and-task. Consider a goal g_i in this list, its siblings to the left were completed during an and-private phase of the computation and are thus left as is, uninitialized. The siblings to the right are marked as **READY** with its goal-frame already pushed into the $GF[]$ array.

2. Terminate the publication of and-work by pushing the goal-frames collected in $GF[]$ into the and-agent's run-queue and by updating the and-sentry node $ASE = N_ASE$.

When and-publicizing work one also has the additional choice of marking each goal-slot directly available in the and/or- tree as **TO_BE_LAUNCHED**. The reason for the existence of this type of tag is twofold. First, it allows us to include a builtin mechanism in LGSATS to help control the *and-or speculativeness* of and-or tasks (identified in 3.6); secondly it gives us a mechanism for fast relaunch of goals after backtracking.

7.10 Memory Organization in Detail.

We have already seen that and-parallelism can be visualized as a process in which the work needed to produce a complete computation branch of the Prolog SLD resolution tree is broken up into smaller sub-trees, some of which are computed ahead. As soon as a sub-tree completes they must be incorporated into the resolution tree respecting Prolog order. Each and-task address space is thus organized in the following way:

- Each and-task is made up of a linked list of *and-task memory segments*.

- Each *and-task memory segment* consists of a contiguous chunk of memory from each engine stack, namely Trail, Heap, Environment, and Choice-Point Stack.
- A special choice-point, the *header*, indicates the start of a new memory segment. The header may be a T-node or an L-node. The lower physical limits for each memory chunk are saved into the header.

The segment header will also record the physical ends of an and-task segment which are stored when **(i)** we reach the end of an and-task; **(ii)** we suspend the execution of and-task; or when **(iii)** the and-agent proceeds execution after a successful backtracking operation in a new and-task.

- A T-node always indicates the start of an and-task. As a consequence of and-task *entrapment* that results from a previous backtracking or suspension operations, an and-task may include memory segments linked together through L-nodes. To determine the physical top of a completed and-task we need only to follow this linked list to reach the first segment, that is the T-node of the and-task.

Memory is allocated as follows:

- An and-agent can only allocate a new memory segment within its stack-set.
- Each and-agent contains a linked list of its allocated memory segments. It is composed by all the and-task memory segments the and-agent has allocated within its stack-set. Some segments may be marked as freed, but space will only be recovered when there are no other segments above. Observe that in this way memory holes can develop during execution.
- The *PASH* register indicates the previously allocated *segment header*, *SH* that must be top-most in the allocation stack. Whenever an and-agent allocates a new memory segment it checks to see if the memory segment pointed out by *PASH* is still in use. If this is not the case then it follows its physical linked list until it finds the first memory segment that is still in use. A new memory segment is then allocated immediately after the end of this last segment: **(i)** all and-agent registers that refer to the top of stacks are updated; **(ii)** the *PASH* register is set to point to the new segment; **(iii)** the new open segment is placed at the end of the and-agent's doubly linked list of memory segments.

Figure 7.9 shows an example of how T-nodes can be used as segment headers for a two processor execution of the following program:

$a(X) \quad :- \quad b(X) \quad \& \quad c(X) .$

$c(X) \quad :- \quad d(X) \quad \& \quad e(X) .$

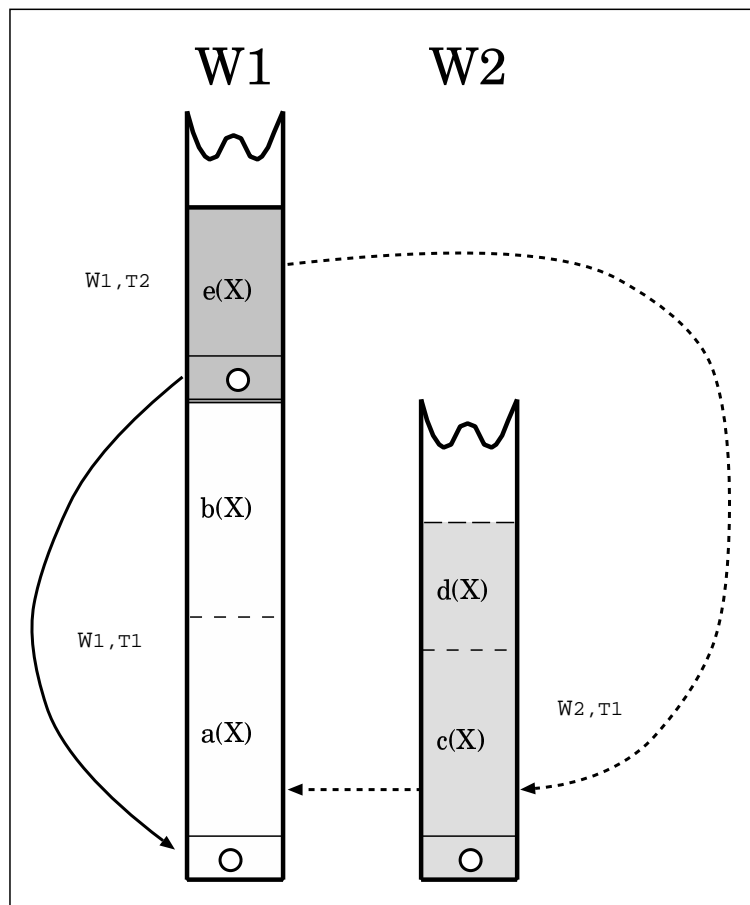


Figure 7.9: Tasks memory organization.

We assume that procedures for $b/1$, $d/1$, and $e/1$ are sequential. Worker W_1 executes $a(X)$, $b(X)$ and then $e(X)$. Worker W_2 executes $c(X)$ and $d(X)$. The figure only shows the choice-point stacks. Each grayed box correspond to a different and-task. Each little circle represents the and-task's T-node. The first and-task is labeled with $W_{1,T1}$ but in practice it could have been created by W_1 long before executing $a(X)$. The second and-task is labeled by $W_{2,T1}$ and actually has its origin at $e(X)$. Worker W_2 has a single and-task. The contiguous line shows the physical pointers in the stacks for W_1 . These pointers are used to reclaim space: whenever the stacks for a task can be deleted we remove its SH for this linked list, if the SH had another SH above, or we physically recover the space, if this one was the top SH.

The dotted lines represent the *implicit* Prolog left-to-right connection order between tasks that are brought into existence by the and-tasks *origin*. Remember that the and-tasks *origin* is the goal-slot of the goal that started the and-task. Whenever the goal to the left of the *origin* is terminated there comes into being an implicit connection from this and-task and the and-task that terminated the goal that sits immediately to the left. Prolog order among and-tasks is therefore implicitly maintained by the and-tasks *origin* goals relative positions within their parallel-environments.

Our design does not make this linkage explicit because we do not want to impair forward execution with redundant information that can be *lazily* obtained from the and/or- tree when needed, namely when backtracking or when determining whether a task is or- leftmost within the and/or- tree.

7.11 Summary.

In this chapter we have presented a novel execution model and runtime system (LGSATS) for IAP Prolog execution complete with data structures, new machine registers and WAM like instructions. The main contributions are coarser representation for IAP computations through and-tasks and small overhead during forward execution, even in the presence of parallel constructs. Also, because LGSATS copes with just small adjustments into the data structures already present in the WAM, implementation of LGSATS based on already optimized WAM based systems should be much simpler to realize than the previous presented models.

In the next chapter we take a fresh look at how to backtrack in IAP Prolog systems. We will see that by changing focus from *parcalls* to *choice-points* as the central point in IAP backtracking, and by using LGSATS as a departing point, we can obtain a simpler mechanism that can allow us to exploit orthogonality better.

Chapter 8

Backtracking under IAP Revisited.

It is widely recognized that backtracking is the most complex operation in Hermenegildo’s classical model for IAP [75]. Experience gained by the implementation of systems such as the &-Prolog [74], DASWAM [124], and &ACE [105] shows that backtracking in IAP is very complex and extremely difficult to implement correctly and efficiently. In our own experience, the complexities of parallel backtracking are particularly troublesome for And/Or Parallel Systems.

In this chapter we take a fresh look at this issue by taking advantage of the new runtime execution behaviour provided by LGSATS. We present what we argue is a simpler and hopefully more efficient design for backtracking in the presence of IAP that preserves Prolog semantics. In a first instance we will ignore the presence of side-effects, and only consider pure Prolog code in our discussion.

8.1 Backtracking in LGSATS.

Classical models for IAP perform backtracking parcall by parcall. The parcall is therefore the “building block” element for backtracking. One consequence is fine-grained backtracking operations, as each backtracked parcall becomes a contention point between workers that cannot proceed up in the and-tree until all computation below the parcall are undone.

By contrast, our basic “building block” element for backtracking will be the and-task (section 7.2.1) – a coarser grain-size object. As we shall see next, the contention point where we will have to wait for the relevant computations to be undone is the *choice*-

point to where we have to backtrack to. Our scheme thus provides a natural extension to sequential Prolog.

The LGSATS backtracking algorithm basically works in the following way. Suppose and-agent w_i fails to choice-point B, we can recognise three different cases: (i) *private* backtracking; (ii) *public* backtracking within the *same* and-task; and (iii) *public* backtracking to a *different* and-task. We describe each case in detail.

8.1.1 Private Backtracking

If B is local to w_i 's current and-task and $MRL(B) > iap_level(ASE)$ then backtracking falls within the and-private part of the and/or- tree. We would like for backtracking to proceed essentially as in sequential Prolog. Moreover, we may want to take advantage of the independence between parallel goals to perform intelligent backtracking, whenever possible.

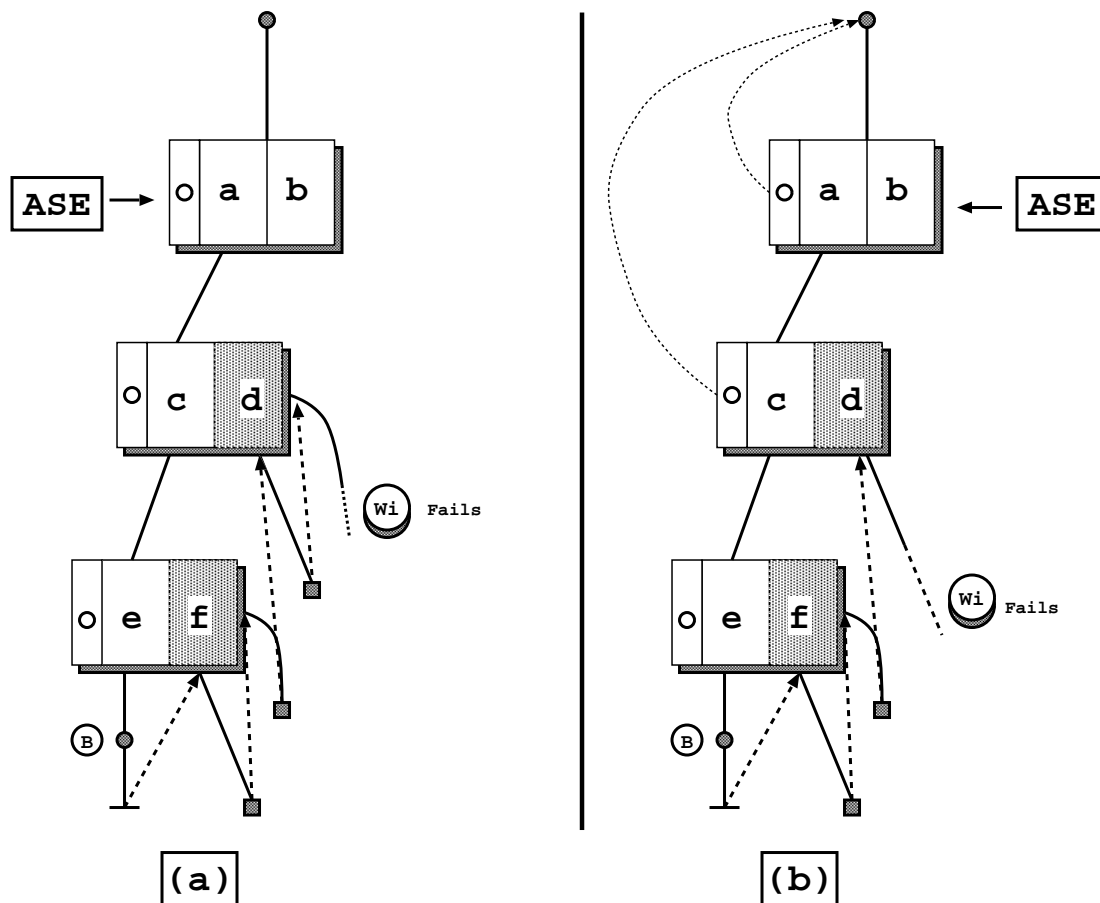


Figure 8.1: And-private IAP backtracking.

Figure 8.1.(a) shows the and-sentry environment pointing to parallel environment $(a|b)$ and worker w_i positioned in the continuation of $(c|d)$, backtracking towards **B**. Prior to this operation, w_i was executing the continuation of **d** and had bypassed goal **f**, the continuation of $(e|f)$, goal **d** and the continuation of $(c|d)$.

In this case, and-agent w_i will unwind the computation it did after **B**, *exactly as is done in sequential Prolog*. The highest point in the and/or tree that has been reached by the current and-task (the minimum reached level) for **B** is the *iap-level* for $(c|d)$. This means that $(c|d)$ is the *referee* for this backtracking operation and its status decides on how to proceed after the unwinding. Since, in this case, $(c|d)$ is in *outside* mode, the and-agent resumes forward execution with another alternative of **B** without having to relaunch any goals because $MRL(B) > iapLevel(ASE)$.

Figure 8.1.(b) shows a similar setting but with w_i still under $(c|d)$. In this case, w_i unwinds the computation it did after **B**, exactly as is done in sequential Prolog. Since the backtracking *referee* is still in *inside* mode we can take advantage of the independent nature of the computations to conclude that no matter whatever alternatives are left for execution at **B**, the computation will fail again at the same point. Failure should therefore be propagated towards $(c|d)$'s parent choice-point. This second backtracking operation is no longer and-local because the next choice point is already located in the and-public part of the computation tree

8.1.2 Public Backtracking Within the Same And-Task

If **B** is local to w_i 's current and-task but $MRL(B) \leq iapLevel(ASE)$ this means that there are parallel and-public environments involved in this backtracking operation. We are therefore performing and-public backtracking and care must thus be taken with IAP constructs where some other and-tasks could be anchored or even with parallel environments belonging to some other and-tasks. This is illustrated in figure 8.2.

Figure 8.2.(a) shows the and-sentry environment pointing to parallel environment $(c|d)$ and worker w_i positioned in the continuation of $(a|b)$, and backtracking towards **B**. w_i bypassed goal **f**, continuation of $(e|f)$, goal **d**, continuation $(c|d)$, goal **b** and the continuation of $(a|b)$. In this case, the highest point in the and/or tree that has been reached by the current and-task (the minimum reached level) for **B** is the *iap-level* for $(a|b)$. This means that $(a|b)$ is the *referee* for this backtracking operation and its status decides on how to proceed. Since, in this case, $(a|b)$ is in *outside* mode, we must backtrack to **B**, but the parallel goals **d** and **b** that sit in the tree between **ASE**

8.1.3 Public Backtracking To A Different Task

Lets start by giving a general idea for the definition we intend for the notion *killing-scope*. Generally an and-task or and-agent sits within the killing scope of a choice-point B when they are affected by any backtracking operation targeting B. This is made more precise in section 8.3.1.

If B belongs to another and-task, then the and-agent *publicly fails*. The operation may involve several agents, so we do as follows:

- allocate a *backtracking frame* at B to synchronize all and-agents in order to decide which ones are under the killing scope of B and therefore need to stop their current computation and start failing towards B.
- Send a backtracking message to all the other members of the team.

Our algorithm initially assumes the worst case scenario, where all and-agents need to backtrack. Therefore, on message arrival, all and-agents interrupt their work to verify whether they are under B in the And/Or tree. The and-agents under the scope of B begin to fail. The others just acknowledge the message by reporting back to the backtracking frame and then proceed with their current work.

The and-agents that are failing towards B are named *backtracking companions*. Their task is to completely unwind all and-tasks that are within the killing scope of B. When an and-task is completely undone, its minimum reached level is compared with the minimum value held at the backtracking frame at B. If the new value is smaller, the one in the backtracking frame is replaced. After all and-tasks are undone, the minimum value in the backtracking frame calculated is the iap-level of the highest point reached in the And/Or tree, that is the backtracking **referee** for this operation. We call *piranha mode* (section 8.4) the process in which the backtracking companions are involved.

The last and-agent to leave the piranha mode is the one that will try to resume forward execution.

Figure 8.3 illustrates a particular case of external backtracking. In figure 8.3.(a) we have two and-agents, w_1 and w_2 , and we assume that w_1 starts to fail towards CP_2 . Notice that w_1 's task started at e and goals e , f and g are completed whilst failure is occurring under goal b . All these goals have been bypassed by w_1 into its current task. At this point, w_1 allocates a backtracking frame with a minimum reached level value of

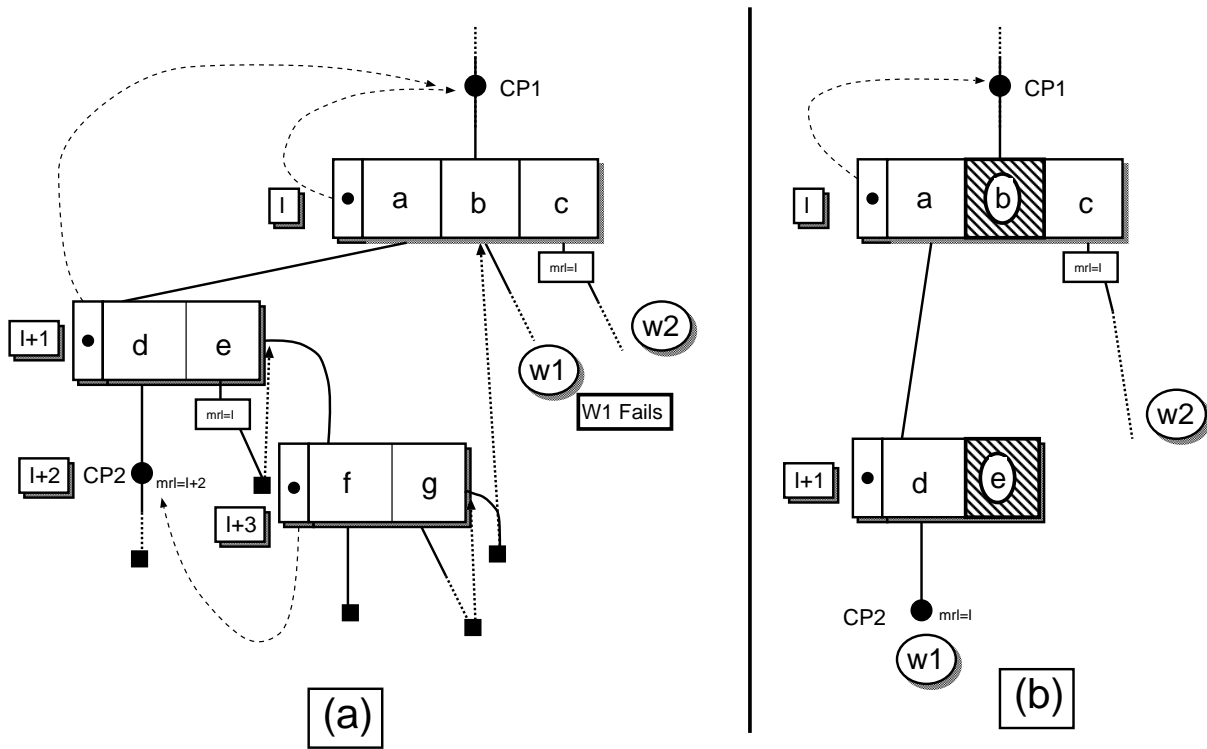


Figure 8.3: IAP external backtracking.

$I+2$ and sends a backtracking message to the other members of the team. As w_2 is not in the scope of CP_2 , only w_1 gets into piranha mode. In 8.3.(b), the figure shows the state of computation immediately after piranha mode. At this point the backtracking frame minimum reached level has been updated to I , w_2 is still computing under c and w_1 is consulting the backtracking referee at I (the minimum reached level). And-agent w_1 has now to decide whether to start forward execution with another alternative of CP_2 or else to initiate a new backtracking operation. Since the backtracking referee $((a|b|c))$ is still in inside mode, w_1 starts a new backtracking operation towards CP_1 . In the sequel, w_1 and w_2 become backtracking companions and the whole tree fails.

In the following sections, we shall refine the LGSATS backtracking algorithm by describing in more detail the backtracking frame data structure and other operations associated with backtracking, such as the piranha mode.

8.2 The Backtracking-Frame

A *backtracking-frame* is a new data structure that is uniquely associated with a choice-point, say \mathcal{C} , whenever the latter is the target of a backtracking operation. It includes six fields:

Choice-Point – a pointer to the backtracking target choice-point B_t .

#Messages-Acks – the number of backtracking messages targeting B_t that were sent and have yet to be acknowledged. Each time a backtracking message is sent this value is incremented. On message arrival the recipient acknowledges the message by decrementing it.

#Backtracking-companions – the number of and-agents that processed a backtracking message targeting B_t and are within the killing scope of the operation. It is initialized with one as the backtracking and-agent that allocates the backtracking frame is included by default. This always represents the number of and-agents entering in the piranha mode of the backtracking operation.

Minimum-Reached Level – a level counter that indicates which node will act as a backtracking referee. Initially it is copied from the current MRL value held in \mathcal{C} .

Task-Stack – stack of pointers to completed and-tasks that need to be undone. This is initially empty. We also denominate this stack the *piranha-sack* for reasons that will be made apparent later.

Bypassed-Fail – a pointer to another backtracking-frame up in the And/Or tree. When non-null it indicates that, while the current backtracking operation was still being deployed, the And/Or subtree rooted in \mathcal{C} got into the killing scope of another backtracking operation further up in the tree.

Backtracking frames are temporary data-structures local to a team. They are valid while the corresponding and-public backtracking operation is in progress. When the backtracking activities terminate, the corresponding backtracking frame is deallocated and becomes available for reuse by future and-public backtracking operation.

8.3 Backtracking Messages

An and-public backtracking operation is done in a special form of execution denominated *piranha* mode. We represent by `piranha(C)` the and-public backtracking operation that is targeting choice-point `C`. The and-agent that initiates a `piranha(C)` operation (the `piranha(C)` *initiator*) begins the operation by marking its current and-task as `BEING_KILLED`. This way the other team members can consult the status of the goal physical and-task owner before picking a foreign goal for execution. And-agents can thus be prevented from picking up goals from a region of the And/Or tree that is in the eminence of being backtracked over.

The backtracking *initiator*) then allocates and initialises a new backtracking-frame for choice-point `C`. It then assumes a worst case scenario and sends a `BM(C)` backtracking message to all the other members its team. These messages initiate a “race” among team agents towards `C`. On message arrival all message recipients will check whether they are in some way affected by the backtracking operation. If they are, then they enter *piranha* execution mode. Otherwise, they just report back to the backtracking frame and proceed with their work.

8.3.1 Reachability

The process of verifying whether an and-agent is within the killing scope of a choice-point `C` is not straightforward. To do so, we present an algorithm that uses the iap-levels that we have presented in the previous chapter (section 7.7). We will assume `C` is the target choice-point, `BM(C)` is the backtracking message, `B` is the and-agent’s current choice-point register, and `anchor[T]` is the parallel environment containing the parallel goal that started the execution of the and-agent’s current and-task (`T`).

An and-agent is within the killing scope of choice-point `C` if `C` is reachable from node `B` in the And/Or tree, that is if `reachable(C, B)` returns `TRUE`. Before outlining the rules for the reachability test algorithm, we first define the notion of *reaching_limit(C)* and of *left-task* as it will simplify and optimise the test.

Definition 14 Reaching-limit for a choice-point `C`: maximum iap-level of a parallel environment belonging to the same and-task and older then `C`. If there is no such environment, iap-level of the task’s T-node.

The reaching-limit is used in a “termination-test” for the *reachability algorithm* pre-

sented next in the algorithm description.

Given a task T , the partial function *left-task* gives the task that precedes T in Prolog order:

Definition 15 T_{i+1} is the left-task to T_i if (i) T_i 's *origin goal* is g_j and T_{i+1} is the task that terminated g_{j-1} ; or if (ii) T_i 's origin goal is the continuation for a parallel conjunction whose last goal is g_j , and T_{i+1} is the task that terminated g_j .

The completion property guarantees that tasks that start from a continuation (case (ii)) always have a left-task. Tasks that origin in and-goals may or not have a left-task.

Definition 16 T_{i+1} is the *strict* left-task to T_i if (i) T_{i+1} is the left-task to T_i and (ii) T_i 's *origin goal* g_j belongs to an already completed parallel environment, eg, *anchor*(T_i) is in outside status.

The algorithm *Reachability Test* is as follows:

- if B and C belong to the same task, then
 - if $C \leq B$, then return YES meaning that C is reachable from B;
 - Otherwise return NO.
- if B and C do not belong to the same task, then
 - consult the T-node T_B of the and-task that owns B, and
 - let PE be the T_B 's anchor (parallel environment from where the and-task started), and PE(B) be its parent choice-point;
 - if PE is still not terminated (it is in inside status), then:
 - * if $iap_level(PE(B)) \leq reaching_limit(C)$ then C is not reachable from node B; return NO.
 - * Otherwise return the result of $reachable(C, PE(B))$;

This means that we transfer our search to the reachability test of C from node PE(B) in the and-task that physically owns PE(B).
 - Otherwise, PE is completed (it is in outside status), and therefore T_B is part of a maximal list of completed and-tasks $T_n, T_{n-1}, \dots, T_{i+1}, T_i, \dots, T_B$ where each task T_{i+1} is the *strict* left task-for T_i .

For each task T from the task list $T_B, \dots, T_i, T_{i+1}, \dots, T_n$ check whether C belongs to T :

- * if C belongs to some task T then return YES;
- * if C does not belong to any of those tasks in the list, then T_n as the last member of the maximal list must be anchored at a non-terminated parallel environment.
 - if $iap_level(anchor[T_n](B)) \leq reaching_limit(C)$ then C is not reachable from node B ; return NO.
 - Otherwise return the result of $reachable(C, anchor[T_n](B))$.

That is, we must resume our search for the reachability of C from the ancestor choice-point of the parallel environment anchor for task T_n .

We are now in a position to precisely define the notion of *killing-scope*. An and-agent that receives a backtracking message $BM(C)$ and its current position is *reachable* from C is said to be within the *killing-scope* of the backtracking operation $BM(C)$. An and-task is within the *killing-scope* of a backtracking operation targeting B if this choice-point is *reachable* from the and-task T -node.

After processing the backtracking-message, $BM(C)$, all and-agents within the killing scope of C start the parallel failure procedure by moving into $piranha(C)$ backtracking mode.

8.4 Piranha Backtracking Mode.

In *piranha* mode and-agents travel within the And/Or tree and undo computations that are within the *killing-scope* of the target choice-point C .

The first step taken by an and-agent w_i when entering $piranha(C)$ is to mark its current and-task as BEING_KILLED. This and-task, say T_1 , may be linked to the left to other already completed and-tasks T_2, \dots, T_n where T_n is the and-task that contains the target choice-point C or is a task that does not own C but nevertheless falls within the killing scope of C .

Worker w_i thus has the choice of adding T_2, \dots, T_n to the *piranha-sack*, that is to a shared list of tasks to kill, or it may keep the tasks for himself to undo. Clearly, both choices lead to correct execution, but the former option provides ample opportunity to further parallelize task unwinding.

Parallelism may be increased by using idle and-agents within the team. They should

also position themselves in \mathcal{C} , become part of the the **backtracking-companions**(\mathcal{C}), and participate in the unwinding of and-tasks in the *piranha-sack*. Arguably, this is an and-scheduler decision and as such, further developments of these subject fall out of the scope of this chapter. We mention the possibility here in order to show that our backtracking method provides sufficient flexibility to support several kinds of and-scheduler optimizations.

An and-agent in *piranha*(\mathcal{C}) unwinds a task T in two phases.

8.4.1 Undoing Incomplete Parallel-Environments

In phase one, the and-agent searches all parallel-environments belonging to the task that are still in *inside-status*, that is, not yet completed. In the case where the and-task T contains the target choice-point \mathcal{C} the search only considers parallel environments older then \mathcal{C} . For each such parallel environments, it will process all of its and-public goals. Each one of these goals will be processed according to its state:

- **TO_BE_LAUNCHED** – The goal-frame is marked as **INVALID** in its run-queue.
- **RUNNING** – To cope with this situation each and-agent has a public register **TL** (task-level) with the starting iap-level of their current and-task (-1 if they are executing scheduler code). This register can be read by any member of the team.

Whenever a **RUNNING** goal is found the and-agent looks into the execution state of all the other members of the team, and for all that are not in *piranha*(\mathcal{C}) or **feeding-frenzy**(\mathcal{C}) it determines the set of team and-agents whose public register **TL** has a value greater then the iap-level of the goal's parallel environment. To each one of these and-agents it sends a backtracking message **BM**(\mathcal{C}). Note that this could result in and-agents receiving extra copies of the same **BM**(\mathcal{C}) message. However this does not represent a problem for the and-agents that are in *piranha*(\mathcal{C}) or **feeding-frenzy**(\mathcal{C}) mode because each extra **BM**(\mathcal{C}) only needs to be acknowledged.

For the others, in order to minimise the increased number of costly *reachability* tests that would ensue, every and-agent possesses a *reachability cache* indexed by choice-point address. This cache is initialised each time a new and-task is started and is updated each time a new backtracking message **BM**(\mathcal{C}) is received and there is no entry for \mathcal{C} in the cache yet. This way, an and-agent out of the

killing scope of choice-point \mathbf{C} needs to perform the costly *reachability* test only once for all the copies of each backtracking message $\mathbf{BM}(\mathbf{C})$ that it can end up receiving.

- **TERMINATED** – The and-agent must determine the set of completed tasks (with the exception of the current one) that were used to execute the current goal.

Remember that a goal-slot is composed by three cells. The first one will point to the and-task that started the execution of the goal (task T_a). The second slot will point to the and-task that terminated the goal's execution (task T_b). The third, last gives the value of \mathbf{B} when the goal completed.

And-task T_a opens and and-task T_b closes a doubly linked-list of tasks comprising all the and-tasks that were necessary to completely execute the goal. We name this list the *goal-slot tasks necklace*. All tasks except T_b must be terminated: we say the task necklace is complete if T_b is complete, and alive otherwise.

If the necklace is complete, we traverse the linked list of completed task nodes in order to mark all tasks that are still marked as **COMPLETED** as **TO_BE_KILLED**. All task nodes that change status during the necklace traversing operation are pushed into the *piranha-sack*(C).

If the task necklace is alive, we know that the and-agent working at T_b will sooner or later backtrack and process the necklace, so we do not need to perform further processing.

8.4.2 Untrailing

In a second phase, the and-agent traverses the and-task's trail and undoes all variable bindings therein stored. In the case where the and-task T contains the target choice-point \mathbf{C} , the backtracking operations must stop at the physical top indicated by \mathbf{C} . Otherwise all trail entries in the and-task need to be undone.

8.4.3 Propagating Failure

After having unwound and-task T_c the following operations need to be performed:

- The and-task minimum reached level is compared with the value held in \mathbf{C} 's backtracking-frame *minimum-reached-level*. This field is updated if greater.

At the the end of this backtracking operation we should have computed the minimum iap-level reached by all and-tasks that were within the killing scope of $\mathbf{BM}(\mathbf{C})$.

- If the and-task's anchor is complete then $T_{\mathbf{C}}$ is part of a maximal list of completed and-tasks $T_n, T_{n-1}, \dots, T_{i+1}, T_i, \dots, T_{\mathbf{C}}$ where each task T_{i+1} is the *strict* left task for T_i .

For each task T from the task list $T_{\mathbf{C}}, \dots, T_i, T_{i+1}, \dots, T_n$:

- If T is marked as `TO_BE_KILLED` or `BEING_KILLED` the task either has already been put into the *piranha-sack*(C) or is going to be undone by some other and-agent.
 - if T 's T-node iap-level is higher then \mathbf{C} 's reaching limit then stop searching for and-tasks to undo in the And/Or tree and enter into feeding-frenzy mode.
 - Otherwise mark T 's T-node as `BEING_KILLED` and restart the algorithm by undoing this and-task.
- If the and-task anchor is in inside status the and-agent must verify the state of the task that owns the anchor's parallel environment (*anchor*(*TASK*)). It can be in the following states:
 - `COMPLETED` – If *anchor*(*TASK*) does not contain \mathbf{C} it can be marked as `BEING_KILLED`. The and-agent then positions itself at the physical end of the task, and restarts the algorithm.

If *anchor*(*TASK*) contains \mathbf{C} the and-agent positions itself at the physical end of the task and unwinds the computation until \mathbf{C} is reached.

 - `RUNNING` or `BEING_KILLED` – In these cases *anchor*(*TASK*) is going to or is being undone by some other and-agent. The and-agent can therefore position itself at the \mathbf{C} 's backtracking frame and enter what we call the *feeding-frenzy*(\mathbf{C}) backtracking mode, where its job consists in taking and-tasks from the *piranha-sack*(C) to undo.

Special care must be taken when an and-agent in *piranha*(\mathbf{C}) backtracking mode happens to be unwinding the and-task that contains choice-point \mathbf{C} . This unwinding operation must stop at \mathbf{C} . Afterwards, the and-agent will enter *feeding-frenzy*(\mathbf{C}) backtracking mode. Agents in *feeding-frenzy*(\mathbf{C}) unwind the tasks in the *piranha sack*, as described in the detail in the following section.

8.5 Feeding Frenzy Backtracking Mode.

Responding to backtracking messages is mandatory, Therefore all and-agents that are within the killing scope of `BM(C)` will eventually enter `piranha(C)` backtracking mode. In this mode they travel up the And/Or tree unwinding and-tasks or pushing them into the `piranha-sack(C)`. Eventually they end up positioning themselves at node `C`, and they enter `feeding-frenzy(C)` backtracking mode.

Notice that when all possible backtracking-companions leave `piranha(C)` mode, all and-tasks that are within the killing scope of `BM(C)` will either be undone or will be in the `piranha-sack(C)`.

When in `feeding-frenzy(C)` state, and-agents compete for completed and-tasks from the `piranha-sack(C)`, which they will then undo. Agents stay in this state until the `piranha-sack` is empty.

The main difference between `feeding-frenzy` and `piranha` backtracking modes is that in `feeding-frenzy` mode the and-agent can safely assume that it is undoing complete tasks. It therefore does not need, as happens in `piranha` mode, to:

- Verify whether it is undoing the task that owns the target choice-point.
- Verify, when it reaches a task node, that the anchor's task owner is complete or within the computation scope of some other and-agent.

If we take into account these minor differences, the method to undo a and-task in `feeding-frenzy` or `piranha` are essentially the same.

Eventually all backtracking companions will enter `feeding-frenzy` until emptying the “piranha sack”. After undoing all its and-tasks, each and-agent will try to leave this mode. When leaving `feeding-frenzy` mode, each and-agent must verify if the value held in the backtracking frame's `Bypassed-fail` is non-NULL. If so it points to the backtracking frame of an older choice-point `D`. The and-agent will now enter into `feeding-frenzy((D))` mode.

8.6 The Decision Mode.

If `Bypassed-fail` is NULL and other agents are still in `feeding-frenzy(C)`, the agent can enter the and-scheduler to look for more and-work. The last and-agent to leave

`feeding-frenzy(C)` is called the “*piranha that lost the backtracking-race to C*”. An and-agent knows that it is the last to leave `feeding-frenzy(C)` mode when (i) after decrementing the backtracking frame’s `#Backtracking-companions` it realizes that the field is 0; and (ii) `#Messages-Acks` is also 0, that is, all backtrack messages `BM(C)` have been acknowledged. Under these conditions the and-agent enters the `decision(C)` state. Only one and-agent, the backtracking-race loser, enters this state.

Even if a backtracking-companion decrements `#Backtracking-companions` to 0 the and-agent can only leave the `feeding-frenzy(C)` state if `#Messages-Acks` is 0. If this is not the case some and-agent w_i did not yet acknowledge the arrival of the backtracking message and therefore might still become a backtracking-companion. Forward execution cannot thus be resumed until all backtracking messages are accounted for. If this and-agent w_i becomes a backtracking-companion it can also bring more and-tasks within the killing scope of C to be undone.

To cater for this situation, when an and-agent tries to leave the `feeding-frenzy(C)` state and finds that (i) `#Backtracking-companions` would be decremented to 0 and (ii) there are still some unaccounted backtracking messages (`#Messages-Acks > 0`) then the and-agent enters into the `piranha_sleep(C)` state to wait for the unaccounted messages. It enters again into `feeding-frenzy(C)` mode if the `piranha-sack` becomes non-empty once again as a result of a new backtracking companion being found. If all backtracking messages are accounted without the appearance of new backtracking companions then the and-agent enters into the `decision(C)` state.

Sooner or later one backtracking-companion will end up entering the `decision(C)` state of backtracking. This and-agent is faced with the following situations:

- Backtracking to choice-point C was *REGULAR*.
- Backtracking to choice-point C was an *INSIDE* operation.
- Backtracking to choice-point C was an *OUTSIDE* operation.
- Backtracking to choice-point C fell within the killing scope of a higher choice-point D .

We analyse each case in detail next.

8.7 Transitive Backtracking

We first analyse what happens if a backtracking operation falls within the killing scope of a different backtracking operation and how to cope with it.

First, remember that our backtracking scheme assumes that all and-agents must process all backtracking messages independently of their state of execution at message arrival. Namely, all and-agents in *piranha* or *feeding-frenzy* mode must continue to receive and process backtracking messages.

Second, notice that the property of being within the killing scope of a choice-point is transitive. In other words, if an and-agent w_i is within the killing scope of choice-point C , and choice-point D is reachable from C then and-agent w_i is within the killing scope of D .

When an and-agent w_i initiates a public backtracking operation targeting, say choice-point D it sends a backtracking message $BM(D)$ to all the other members of the team. If the $BM(D)$ recipient is in *piranha*(C) or *feeding-frenzy*(C) mode it tries, on message arrival, to acquire a lock for their current **Bypassed-fail** pointer. The first one of C 's backtracking companion, say w_0 , to acquire this lock is faced with the following possibilities:

1. The **Bypassed-fail** field is NULL. In this case w_0 must check if choice-point C is within the killing scope of choice-point D .

If C is within the killing-scope of D , w_0 updates **Bypassed-fail** to point to D 's backtracking-frame. Otherwise, w_0 updates **Bypassed-fail** with an integer indicating the number of C 's backtracking-companions minus one.

2. The **Bypassed-fail** field points to another backtracking-frame. In this case w_0 finds the *last* backtracking frame LBF in the **Bypassed-fail** linked list and tries to acquire a lock to its **Bypassed-fail**, say LBF(FAIL). When it gets the lock, it has three alternatives:

- (a) LBF(FAIL) is NULL. In this case w_{first} must check if LBF's choice-point is within the killing scope of choice-point D .

If LBF's choice-point is within the killing-scope of D , **Bypassed-fail** w_0 updates it to point to D 's backtracking-frame. Otherwise, it copies the value held in the LBF's **#Messages-Acks** to LBF(FAIL).

- (b) Otherwise, `LBF(FAIL)` must be non-NULL. Since when w_0 first tried to get the lock it was NULL, w_0 was beaten by some other and-agent in the locking race. The and-agent does nothing.
- (c) And-agent w_0 then releases the lock at `LBF(FAIL)`

3. And-agent w_0 releases the lock on `C`'s backtracking-frame `Bypassed-fail` field.

All other `C` backtracking companions are then faced with a `Bypassed-fail` field that either holds an integer or a pointer to another backtracking frame. In the former case, they decrement the integer by one and then release the lock. In the latter case, they just release the lock. The backtracking-companions are then free to continue with its current backtracking activities.

Eventually all workers will leave `piranha` and `feeding-frenzy` mode. If the and-agent that loses the backtracking race finds that the `Bypassed-fail` field is not NULL, then it points to a backtracking-frame for a choice-point `D`, and the worker must continue to fail, this time targeting choice-point `D`. In other words, it becomes a `backtracking-companion(D)` in `piranha(D)` mode. On the other hand, if the current `Bypassed-fail` field is NULL it must determine whether the just finished backtracking operation was `INSIDE`, `OUTSIDE` or `REGULAR`, as discussed in the next section.

If the `Bypassed-fail` field is NULL all the other `C` backtracking companions have to enter the and-scheduler. Otherwise, each and-worker can choose to become one of `D`'s backtracking-companions in state `feeding-frenzy(D)`, or enter into the and-scheduler.

8.8 Inside, Outside and Regular Backtracking.

Classical IAP schemes perform backtracking in a parcall by parcall basis. Backtracking and-agents must verify if each parcall is in inside or outside in order to decide how to perform backtracking.

In contrast, LGSATS performs backtracking and-task by and-task. Each and-task may pass through several parallel environments during its execution. Furthermore, through the computation of the *minimum reached level* we determine the iap-level and, if it exists, the associated parallel environment whose *I/O* status determines whether we are performing an *inside*, *outside* or a *regular* backtracking operation.

If the minimum reached level calculated at the backtracking-frame is greater to or equal to the iap-level of the target choice-point then we are in the presence of a regular backtracking operation: there is no need to relaunch any and-goals or to continue with more backtracking activities.

Otherwise, the computed minimum reached iap-level is strictly smaller then the the iap-level of the target choice-point. The minimum reached level corresponds to the highest point in the And/Or tree reached by all the backtracking-companions during forward execution of and-tasks that were within the killing scope of the target choice-point. The parallel environment with this iap-level is called *backtracking-referee*.

8.8.1 Inside Backtracking.

Next, we concentrate on Prolog computations that do not perform side effects. We show that if the *backtracking-referee* is still in inside-status we can take advantage of the independence of the parallel computations in order to redirect our backtrack operation to the *backtracking-referee*'s parent choice-point.

Suppose that the last backtracking-companion has entered `decision(C)` and that the *backtracking-referee* contains independent goals $G_1, \dots, G_i, \dots, G_n$. There is at least one independent sub-goal G_j that was bypassed into an and-task that fell within C 's killing scope. The iap-level of the *backtracking-referee* is strictly smaller then the iap-level of C 's, but the *backtracking-referee* is still within C 's killing scope. Therefore, C must be located in an and-task in the execution scope of a goal $G_i, i < j$. By the **completion property** this goal G_i must have been completed when goal G_j was bypassed.

The point of failure responsible for the current backtracking operation must thus be located under the computation scope of one goal from the set $\{G_j, G_{j+1}, \dots, G_n\}$ because the *referee* environment is still in inside-status. Lets call this goal G_{j+k} . Computations under the scope of G_{j+k} cannot have created extra choice-points because otherwise we would not have been backtracking towards C . Therefore, goal G_{j+k} must have failed *deterministically*. Since goal G_j is independent from the other goals in the backtracking-referee: *there cannot be a solution to the conjunction*.

We therefore can and should continue to fail, but this time backtracking to the parent choice-point of the *backtracking-referee* ($BR(B)$). We can implement this by making the and-agent that lost the piranha-race initiate a new IAP backtracking operation,

this time targeting $\text{BR}(\text{B})$.

This form of intelligent backtracking can no longer be applied as described in the presence of side-effects, as it does not respect Prolog semantics. In this case we are only allowed to kill tasks in the scope of sub-goals G_l such that $l > j + k$.

8.8.2 Outside Backtracking.

If the *backtracking referee* is completed (in outside mode), failure must have occurred while executing its continuation. When w_l enters $\text{decision}(\text{C})$ state the continuation and some computations within the *backtracking referee* will have been undone, the ones under and to the right of C . This means that some parallel goals may again become available for parallel execution.

Suppose that the backtracking target is the choice-point B and that the *backtracking referee* is the parallel environment BRE . The set of goals that should be relaunched can be found by following the environment chain $B(E) \Rightarrow \dots \Rightarrow \text{BRE}$ and finding out which members of the chain are parallel environments. All and-goals to the left of $P(E(\text{CP}))$ must be made and-public. The process is in fact very similar to making and-work public, as described in section 7.9.

After relaunching these and-goals, the and-agent is then free to execute the next alternative for the target choice-point.

8.8.3 Regular Backtracking.

We are in the presence of a regular backtracking operation when the minimum reached level is greater to or equal then the iap-level of the backtracking target choice-point.

This means that there is no *backtracking referee*. Instead, the situation is similar to Prolog where undone computation is entirely under the scope of the goal that created the target choice-point. The and-agent just needs to check if the `Bypassed-fail` pointer is `NULL`. If so, it can take the target choice-point's alternative

8.8.4 Iap-Levels After Successful Backtracking.

Before resuming forward execution, a worker w_l in `decision(C)` must update several iap-levels:

- Set C 's *minimum reached level* to C 's iap-level.
- Set the *minimum reached level* in the T-node of the and-task that owns C ($TASK(C)$) to $\min(iap - level(anchor(TASK(C))), iap - level(C))$
- Set w_l 's current iap-level to $iap - level(C)$.

8.9 An Example of Inside-Backtracking.

Figure 8.4 presents an example of *inside-backtracking*. We have three workers, w_0 , w_1 and w_2 , and three tasks, T_a , T_x and T_b . All parallel environments have been created by the same and-task (T_a). The task also owns choice-points C_1 and C_2 . Task T_a is complete, because and-agent W_1 started task T_x at goal g before task T_a could bypass it.

The key for this example is how T_x starts from the bottommost goal g and then bypasses and-goals e and b from parallel-environments with successive smaller iap-levels. We assume that task T_a was already closed when task T_x reports success to goal g . Thus, the continuation of the bottommost parallel environment belongs to T_x .

When and-agent W_1 fails in task T_x the task's *minimum reached level* is iap-level i . Notice that T_x had started at iap-level $i+2$, the iap-level of the parallel environment containing goals f and g . Worker W_1 B register points to choice-point C_2 . This is so because during completion of the parallel environment $(f|g)$ (at iap-level $i+2$) W_1 determines that the deepest rightmost node (DRM) of the parallel environment is C_2 . It thus set B to this choice-point before bypassing the continuation. As execution of task T_x by W_1 is deterministic until it fails, there were no further modifications to the and-agent's B register. Also notice that, since w_1 's B register always points to C_2 , whenever task T_x moves up in the and/or- tree, C_2 's minimum reached level is updated accordingly.

Also notice that after bypassing goal b into task T_x , but before W_1 would fail, and-agent W_2 grabbed goal c and started the execution of a new task T_b . So when W_1 fails two agents, W_1 and W_2 are active in this sub-tree, whereas W_0 has completed task T_a .

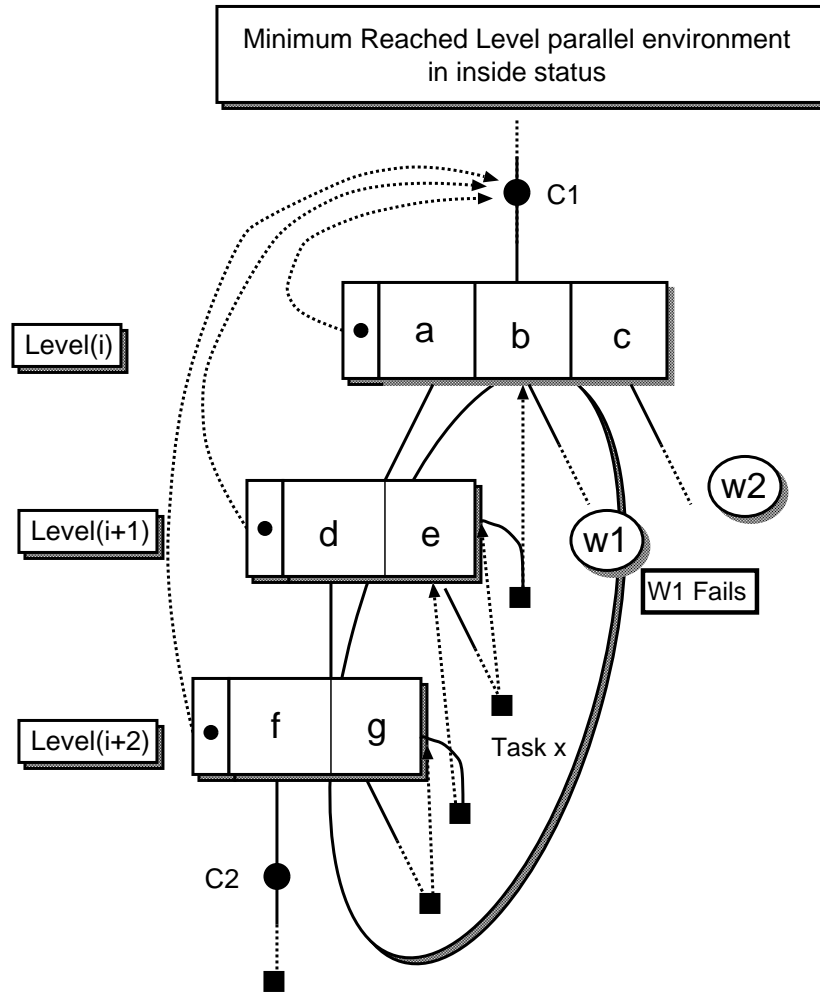


Figure 8.4: Detecting an INSIDE and REGULAR backtracking operations.

Next we describe how our algorithm implements W_1 's failure. First, W_1 allocates a backtracking frame in node C_1 . It then sends a backtracking message BMC_2 to the other members of the team.

And-agents W_0, W_3, \dots, W_n , are out of C_2 's killing-scope. Hence their response to the message is just to decrease $\#Messages-Acks$ in the backtracking frame.

When and-agent W_2 receives the backtracking message it determines that task T_b is *anchored* at the incomplete parallel-environment that owns goal a and goal b .

By following our reachability algorithm (section 8.3.1) the search for the killing scope is then transferred into the and-task that owns the parallel environment parent's choice-point. The reachability search is thus resumed at this node, which happens to be C_1 at iap-level $Level(i-1)$. This iap-level is strictly smaller than the iap-level of node

C_2 , therefore task T_b does not sit within the killing scope of the broadcast message. And-agent W_2 can thus safely decrement $\#Messages-Acks$ in C 's backtracking frame and resume forward execution.

And-agent W_1 enters into `piranha(C_2)` mode after emitting the backtracking message. When W_1 finishes undoing task T_x it consults the task node and finds that the completed task to the left is task T_a . Task T_a owns C_2 , so this falls in the case where an and-agent in `piranha` mode must stop backtracking at C_2 instead of undoing the entire task T_a .

The `piranha-sack` at C_2 is empty, therefore there is no feeding-frenzy activities for and-agent W_1 . When W_1 decrements $\#Backtracking-companions$ it reaches 0.

Since $\#Messages-Acks$ is 0, and-agent W_1 realizes that it has lost the `piranha-race` with the computed minimum reached-level `Level(i)`. The backtracking referee is thus the parallel environment where goal `a`, `b` and `c` are located. The parallel referee is in `inside-status`, therefore backtracking must be propagated to its parent choice-point, node C_1 . This is achieved in the form of a second failure operation whose target is now C_1 . The operation again starts by sending a backtracking message `BB(C_1)` to all the other team members after which and-agent W_1 enters `piranha(C_1)` mode.

Imagine W_2 is still computing goal `c` (e.g., still within task T_b). When W_2 receives the message it realizes that it is sitting within the killing scope of this new backtracking message. It thus enters into `piranha(C_1)` mode and undoes all the computations it has done within the scope of goal `c`. Notice that if in the meantime other and-public goals from task T_b had been taken by other and-agents, these tasks and their executing agents would also fall within the killing scope of C_1 . Eventually the whole task T_b will be undone and forward execution resumed, by whichever and-agent that lost the backtracking race, through the execution of the next alternative from node C_1 .

In the case where goal `c` was *terminated* when W_2 received the backtracking message, W_2 must be executing a different task. When and-agent W_1 undoes computations towards node C_1 , W_1 also checks all incomplete parallel environments that sit between C_1 and C_2 for and-tasks in the completed public goals *task-necklaces* that are marked as `COMPLETED`. The and-agent marks these and-tasks as `TO_BE_KILLED` and puts a reference to them into the `piranha-sack`. Goal `c`'s task-necklace is therefore searched and all its component and-tasks placed at the `piranha sack`.

These tasks are then undone by W_1 while in `feeding-frenzy(C_1)` mode. Eventually there will be no more tasks to undo and an alternative from node C_1 can be taken.

A new LINK node for task T_a is allocated and forward execution resumed at level `Level(i-1)` by and-agent W_1 , although still within the computation scope of and-task T_a , the task that owns node C_1 .

8.10 An Example of Regular-Backtracking.

The last example (illustrated by figure 8.4) consists of two first backtracking steps: first to C_2 , and then to C_1 .

When backtracking from C_2 to C_1 the minimum reached level is the iap-level of choice-point C_1 . We are thus in the presence of a *regular backtracking* operation, hence the and-agent that loses the backtracking race towards C_1 is able to immediately resume forward execution with the next alternative from the node if its backtracking-frame's `Bypassed-fail` pointer is `NULL`.

8.11 An Example of Outside-Backtracking.

Figure 8.5 presents an example of *outside-backtracking*. In this example and-agent W_1 was expanding task T_f while and-agent W_2 was expanding task T_g anchored at $(h|i)$. The other and-tasks, T_a , T_b , T_c , T_d and T_e , are already terminated. And-task T_a is the physical owner of parallel environments $(f|g)$, $(c|d)$ and $(a|b)$. And-task T_c originated at the continuation of $(f|g)$ and managed to move up in the tree when it bypassed goal d . And tasks T_b and T_e fall completely within the scope of execution of its goal origin. And-task T_d started with the continuation of $(c|d)$ but is not able to move up in the And/Or tree because when, say and-agent W_1 executes b 's `check_par` instruction the goal has already been taken to start and-task T_e , that at this point is already terminated, because the and-agent that was executing it is not able to proceed with the continuation.

When and-agent W_1 reaches $(a|b)$ `close_pcall` instruction in and-search mode it must close the parallel environment. It thus determines that the last choice-point in Prolog order (the `DRM`) within the now completed parallel environment is C_1 . C_1 's minimum reached level field is therefore updated with `Level(i)` and $(a|b)$ status is set to `outside`.

Worker W_1 then initiates task T_f with the continuation of $(a|b)$. W_1 's current iap-level

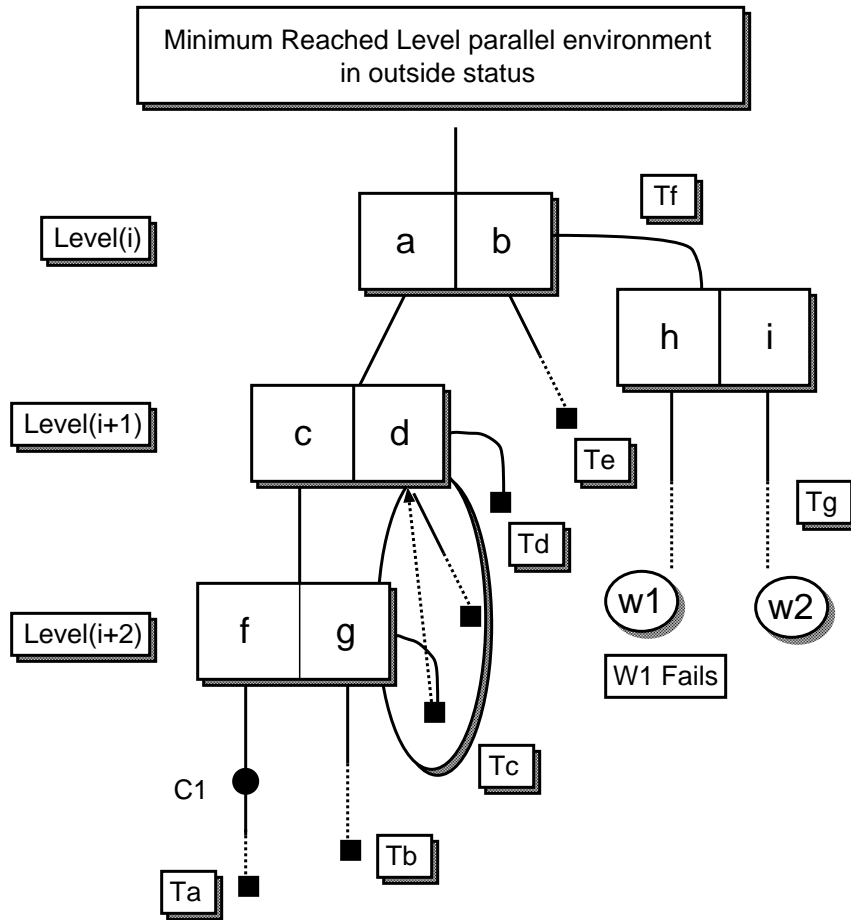


Figure 8.5: Detecting an OUTSIDE backtracking operation.

was set to the maximum of the deepest iap-levels reached during $(a|b)$'s computation. Its B register is set to point to the complete parallel environment DRM, the choice-point C_1 .

Eventually, when and-agent W_1 fails within and-task T_f its B register is still pointing at C_1 resulting in a backtracking message $BM(C_1)$ being sent to all the other team members and the start of a $piranha(C_1)$ backtracking operation by W_1 .

When and-agent W_2 receives the backtracking message it first verifies whether its current task T_g sits within the killing scope of choice-point C_1 . Since the deterministic task T_g is anchored at parallel environment $(h|i)$, W_2 's B register points directly to C_1 . The and-agent thus immediately concludes that it sits within the killing scope of C_1 and enters $piranha(C_1)$ mode.

Worker W_2 next undoes T_g . The minimum reached level at T_g 's T-node is greater than the minimum reached level at the backtracking frame because T_g sits in the

continuation of $(a|b)$. Therefore there is no need to update the backtracking frame's minimum reached level.

And-task T_g is not *strictly linked* to the left and and-task T_f , owner of the parallel environment $(h|i)$, is not in the COMPLETED state. This means that and-agent W_2 's can leave **piranha**(C_1) mode and move to **feeding_frenzy**(C_1) mode.

Meanwhile, W_1 undoes task T_f and reaches its T-node. Task T_f is left-linked to task T_g in the stricter sense, so and-agent W_1 transfers its **piranha**(C_1) backtracking activities to this task. Notice this linkage to the left is the start of the linked list of completed and-tasks T_e, T_d, T_c and T_b .

And-agent W_1 can consult the backtracking-frame to verify whether there are still some other backtracking companions to help in the undoing of this linked list of and-tasks. In this case there is W_2 so W_1 can add tasks to the piranha-sack.

Special care however must be taken with task T_b that owns choice-point C_1 . This task must be undone by and-agent W_1 until node C_1 is reached. At this point the and-agent also enters **feeding_frenzy**(C_1) mode and collaborates with W_2 to undo the remaining tasks in parallel.

We conclude by discussing how to distribute and-tasks to the piranha-sack while backtracking. Notice that and-agent W_2 can enter **feeding_frenzy**(C_1) even when there are no and-tasks in the piranha-sack to be undone. The and-agent cannot know beforehand whether and which and-tasks will be put into the piranha-sack. It all depends on the decisions made by the backtracking companions and on how many completed and-tasks these and-agents are able to find before they position themselves at choice-point C_1 in feeding_frenzy mode. One alternative is for W_2 to wait for a while and then leave the backtracking companions if no backtracking work is forthcoming.

We can use several parameters to tune the algorithm that decides whether and how many tasks from a linked list of completed tasks T_1, \dots, T_n should be added to the piranha-sack. Example parameters include the number of the current backtracking companions, the size of the list, and machine speed, just to name a few. Optimal values for these parameters should be obtainable from application analysis.

8.12 Memory Reuse by Reference Counting.

The final issue we will address in our design for backtracking is how and-agents may attempt to actually recover physical space. Remember that in section 7.10 we remarked that all and-tasks are formed by a linked list of stacks segments, each one with a *header* to determine its physical boundaries.

In LGSATS a segment header can be a T-node or a L-node. Each one of these headers contains a physical pointer to the stack set segment header that physically precedes it in allocation order. Remember that only the and-agent owner of the stack set can allocate memory segments within its stack set. Although other and-agents can logically deallocate those memory segments, effective memory reuse can only be performed by the and-agent that physically owns that space.

Note that a segment's physical predecessor can belong to the same or to a different and-task. It can be in use or be logically deallocated. The pointers that link stack segments are designed to maintain allocation order for the stack segments that have been allocated at each and-agent stack set.

Our memory organization is in fact very similar to the *cactus stacks* used in shared memory ORP systems, particularly in Aurora [26]. In a cactus stack allocation scheme, allocated stack segments can be logically freed by any worker, but only when its space is not going to be referenced again by any worker in the system. The actual space used by a logically freed memory segment, however, can only be reused when all memory segments on top of it are also logically freed.

Figure 8.6 illustrates a typical memory allocation situation for LGSATS. Observe that there are five memory segments S_1, S_2, S_3, S_4, S_5 . Segments S_1, S_2 belong to the same and-task, segment S_2 header is a L-node. Segments S_3 and S_4 belong to and-tasks anchored at a parallel environment $((a|b|c))$ that physically belongs to segment S_1 . Segment S_5 initiates an and-task anchored at a parallel environment $((d|e|f))$ physically belonging to segment S_4 while segment S_6 initiates an and-task anchored at a parallel environment $((g|h|i))$ physically belonging to segment S_2 .

And-agent W_1 is expanding segment S_2 while and-agent W_2 expands segment S_6 . Now suppose that both W_1 and W_2 become backtracking companions and need to undo their current and-tasks. Let us suppose that parallel environment $(g|h|i)$ is incomplete and that W_1 is able to deallocate segment S_2 very quickly, even before and-agent W_2 starts to unwind the trail in segment S_6 . In this scenario it would be possible for segment

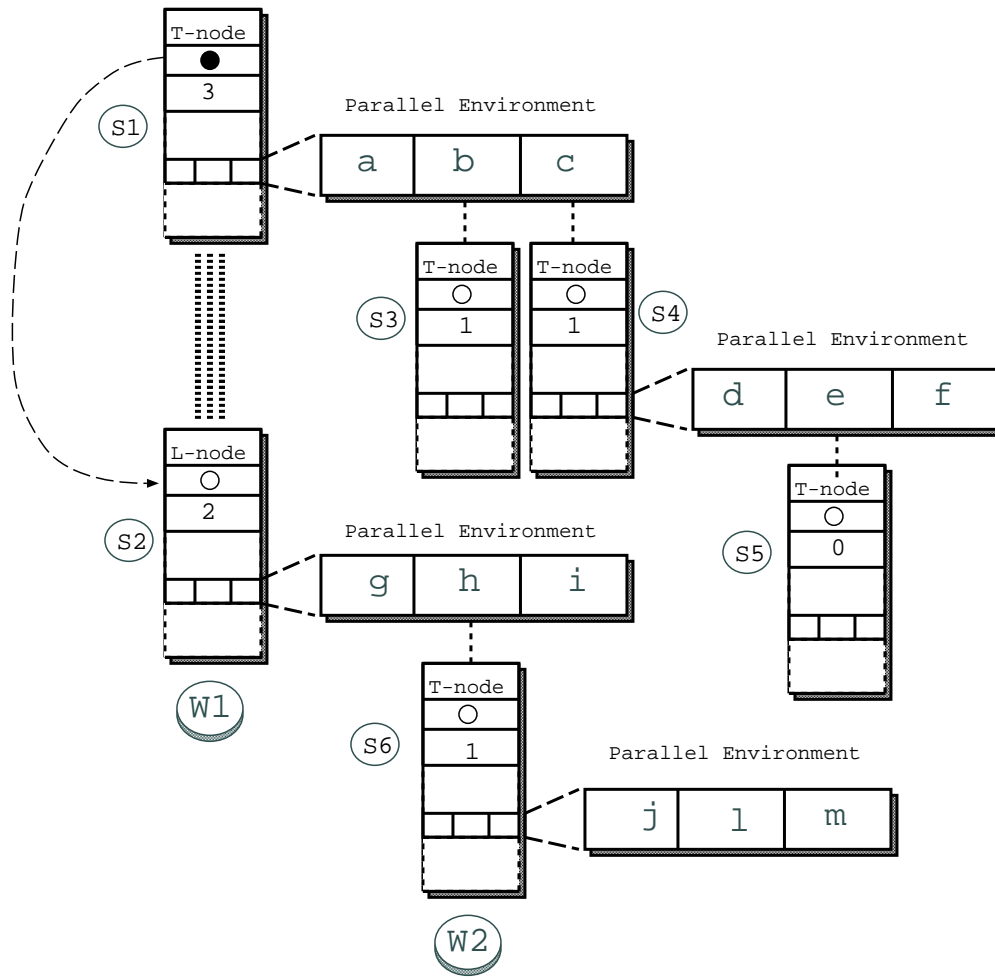


Figure 8.6: Reference counting for memory management.

S_2 to be reused for some other and-task before and-agent W_2 had the possibility to consult parallel environment $(g|h|i)$ as required by our parallel backtracking algorithm. This is wrong and would most certainly lead to incorrect execution.

In general we need to prevent the reuse of stack segments, even when already backtracked, if they contain data structures that still need to be consulted from other stack segments. We can fortunately take advantage of a variation of the well known technique of *reference counting* [95] to cope with this situation.

In figure 8.6 each segment header contains a reference counter that gives the number of other memory segments from where it is possible to access the stack segment during a *piranha* backtracking operation. The main idea beyond this reference counting is that a memory segment is considered deallocated and able to be reused by a different other and-task only when this number reaches 0.

The reference count for stack segment S_1 is 3 because it still has not been backtracked over, this counts as one reference and the two and-tasks that contain stack segments S_3 and S_4 are anchored in parallel environment $(a|b|c)$. This counts as two. All the other reference counts can be analogously deduced. Note that each segment has its own reference count. This is clearly illustrated by segment S_2 whose segment header is a L-node. We finish our example by noticing segment S_5 which has a reference counting of 0 and can therefore have its space reused in the allocation of a new stack segment by some other and-task.

The general rules for stack segments *backtrack reference counting* are therefore as follows:

- During Forward Execution:
 - Each time a new stack segment is allocated its reference counter is initialized as 1.
 - Each time a new and-task is started, the backtrack reference counter of the stack segment that physically owns the and-task anchor is incremented by 1.
- During Backtracking:
 - Each time a stack segment is backtracked over in piranha mode its reference counter is decremented by 1.
 - Each time an and-task is put into the backtracking piranha-sack the stack segment that physically owns the and-task anchor is decremented by 1.
 - Each time an and-agent backtracks over a T-node the reference counter of the stack segment that physically owns the and-task anchor is decremented by 1.

It follows from our rules that backtrack reference counting is “propagated” into other segment headers through the allocation of new T-node stack segments. The allocation of a L-node stack segment does not affect the backtrack reference counting of other stack segments.

8.13 Summary.

We argue that the widely known RAP-WAM backtracking scheme [75], as implemented in most IAP Prolog systems, is not suitable for And/Or Parallel systems because it is rather too complex. We believe that backtracking is one of the main features of Prolog, and that its implementation is of key importance in IAP systems.

Our work on LGSATS has allowed us to address IAP backtracking from a new perspective. We show that by moving our attention from the parallel environment to the choice-point, thus following the example of ORP and sequential prologs, we can take advantage of several properties of LGSATS and-tasks to devise a new iap-backtracking mechanism that we believe will be simpler to implement, more efficient and at the same time easier to integrate in a combined IAP/ORP system. We have also tackled the difficult subject of memory recovery and reuse in IAP backtracking through the use of backtracking reference counters.

Chapter 9

Conclusions and further work

Logic Programming, and namely Prolog, have been used to develop complex programs in areas such as Expert Systems, Machine Learning, and Decision Systems. These programs often take several days, if not weeks, to execute. Therefore, modest speedups achievable through a parallel Prolog system can be directly translated to very significant gains in execution time.

Prolog programs exhibit several forms of implicit parallelism. Depending on the application, one may be dominant or several may be available. Unfortunately, previous work on parallel Prolog only allowed for the efficient exploitation of a single form of parallelism per system, thus only supporting a subset of all applications. The problem of efficiently integrating IAP/ORP for full Prolog programs remained open.

9.1 Contributions

In this thesis we have identified and described in detail subtle difficulties in realizing in practice an efficient implementation of combined And/Or parallel schemes, such as the PBA [64] and ACE [59]. In order to circumvent those problems, we proposed and justified the use of two new IAP/ORP integration design principles, *orthogonality* and *Memory Independence*. We then developed a new environment representation scheme for or-parallelism, the SBA, that is compliant with our design principles and capable of supporting the simultaneous exploitation of of And/Or parallel execution.

As a concept validation experiment, we took the Aurora [91] or-parallel Prolog system and adapted its or-parallel binding representation scheme, a classic binding array, to

the SBA. We concluded that the necessary system changes were reasonably small and very localized. Moreover, the system performed well with very little degradation and often with speed improvements, when compared to the original binding array based system.

By abiding to general rules for good software design we were able to devise the data structures required to fully exploit the full potential inherent in the combination of IAP/ORP in Prolog and provide integration mechanisms for a fully working system. System specification fully supports backtracking with memory reuse and should prove much easier to integrate into an existing SBA based or-parallel system then previous proposals.

9.1.1 The Orthogonality Principle

During our research it became clear that we should follow a major design principle, *orthogonality*: or-parallel execution should be unaware of independent and-parallel execution, and independent and-parallel execution should be able to ignore, as much as possible, the very existence of or-parallelism in the system.

It is normal practice in scientific and engineering endeavors to take advantage of techniques that have been used and proven effective in the past. Orthogonality is required if we want to apply this principle in the design and development of a combined Prolog parallel system. In our case, the two base components will be an and-parallel and an or-parallel system.

9.1.2 Memory Independence

In studying some of the various models described in the literature for combining or-iap parallelism (composition tree + stack copying or composition tree + paged binding array) [62, 59, 64] for full Prolog, we verified (section 4.3) that when we consider the simultaneous execution of both forms of parallelism, the workers were no longer independent of each other in terms of what could happen in terms of private and-parallel executions.

We say that teams are *memory independent* when the way memory is allocated within a team can *never restrict* the way another team is allowed to allocate memory. If memory allocation within a team depends on other teams, and-parallelism cannot be

orthogonal to or-parallelism. We have seen that if this is the case then either the or-parallelism we have available depends on how and-parallelism has been exploited in the other teams, or we will have to undo and-work within the team to guarantee correct execution of or-parallelism. Therefore memory independence is a property necessary to achieve orthogonality.

9.1.3 The Sparse Binding Array

A major result of our work is a *new* environment representation scheme, the *Sparse Binding Array* (SBA), which guarantees memory independence. We demonstrate through experiments that the SBA can perform as well, and often better, than traditional BA based schemes in the presence of ORP, whilst allowing for orthogonality between independent and-parallelism and or-parallelism.

9.1.4 LGSATS

We have designed a completely new IAP execution model, complete with backtracking and memory reuse, which incurs minimum overhead while executing non-public and-parallel code and designed from the beginning to comply as much as possible with the orthogonal integration of iap/or- parallelism.

We start from a standard WAM based sequential Prolog implementation where we incorporate the minimal number of changes that we believe are required for an efficient implementation of IAP. Ultimately, we aim at a novel IAP proposal that can be naturally extended to support ORP through the SBA [39] or other approaches, such as the α COWL [111].

Forward Execution

The key ideas present in our design are: **(i)** to always take advantage of the analogy between ORP and IAP [103]; **(ii)** to avoid creating new structures by re-utilizing WAM data structures wherever possible; and **(iii)** to avoid major changes to the compiler. We shall assume that the underlying system is WAM based [151], like YAP [42, 112].

The new model is called LGSATS, (*Large Grain Sequential And Tasks for SBA*), and its design is based on the following main guidelines:

- And-agents perform as sequential engines for the most part of their execution;
- Follow a lazy work publishing strategy to induce large grain and-tasks for parallel execution;
- Avoid creating new structures (or markers) by re-utilizing WAM data-structures wherever possible. This simplifies system implementation, facilitates iap/or system integration and reduces overheads.
- Avoid major changes to the compiler so that reuse of previous work becomes possible and desirable.

Backtracking

The classical iap- execution backtracking mechanism as proposed by Hermenegildo [75] is very complex and extremely difficult to correctly implement in an efficient way. This has been verified in practice with, for example, the experience gained by the implementation of &ACE [59].

We took a fresh look at this issue and were able to take advantage of the new runtime execution behaviour provided by LGSATS to come up with a simpler and hopefully more efficient way of performing IAP-backtracking operations that preserve Prolog recomputation semantics.

9.2 Further Work

We are continuing research on the data structures for supporting combined parallelism. We have demonstrated that the sparse binding array offers a very clean solution for recomputation based systems. In the future we plan to investigate whether the extra complexity of more space efficient solutions is worthwhile, by researching alternatives such as the shared paged binding array [66], or by using approaches that compress the sparse binding array, such as replacing the array by a fully-associative cache. Our work was used in the design of a copying-based solution that achieves memory independence, the α -COWL [111].

Meanwhile and directly related to what has been discussed in this thesis, in the near future we would like to:

- Produce a working implementation of LGSATS in YapOr and make the Yap compiler generate LGSATS code directly from Prolog code annotated with CGEs.
- Perform extensive benchmarks of the system in shared memory machines with 4-,8- and if possible 16-CPU's in order to compare the results with the other existing optimized IAP systems such as DDAS and &ACE.
- Integrate IA/Or parallelism within the LGSATS/YapOr framework by following the orthogonal integration mechanisms discussed in the thesis.
- Introduce support for the exploitation of side effects into our backtracking mechanism. This should be facilitated by the C-tree nature of our execution model which allows us to decompose the leftmost search into *and-leftmost* and *or-leftmost* searches [65].
- After having a running implementation of LGSATS/YapOr with the SBA we intend to thoroughly study the system memory reference behavior. This can be achieved by running a port of the system in parallel machine simulators [98]. We should also be able to compare the system memory behaviour with existing IAP and ORP systems to assess the real impact that the SBA has in systems performance.
- Study the impact the simultaneous exploitation of IAP/OR has on the performance of real large Prolog applications like Expert Systems, Machine Learning, and Decision Systems where logic programming and in particular Prolog play an important role
- We also plan to use the LGSATS within the DAOS framework to implement scalable and/or parallelism within a distributed architecture. This should prove to be more feasible than the currently proposed DAOS [27] scheme that uses the RAP-WAM to explore ANDP and should therefore prove to be much more sensitive to small granularity issues.

9.3 Final Remarks

I could not finish without this simple remark. We are all tempted to classify earlier work as incomplete, wrong, too simple, too complex or even inadequate. A multitude of adjectives to forget one obvious truth. Science is made from trial and error. Only by climbing on the shoulders of giants, just to paraphrase the old Newtonian cliché,

can we have some hope of widening our horizons. It does not matter that someone's proposal was incomplete or even in some circumstances erroneous. Sometimes just the fact of being wrong is a contribution. Good science is made by finding out where the wrong is and wonder about new ways of fixing it.

When I started working in this area there were many proposals that claimed to be the solution to the IAP/ORP integration in Prolog. Time eventually proved them wrong. This work is obviously not the ultimate solution, as none can ever be. My hope is that when someone reads these lines he had already started to think on how to improve on my ideas and even find some unexpected situations where my proposals fail. If this happens, it means that I have succeeded because someone else has started to improve upon what has already been done.

References

- [1] Hassan Aït-Kaci. *Warren's Abstract Machine — A Tutorial Reconstruction*. MIT Press, 1991.
- [2] Khayri A. M. Ali. Or-parallel Execution of Prolog on the BC-Machine. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 253–268. MIT Press, 1988.
- [3] Khayri A. M. Ali and Roland Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *Proceedings of the North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.
- [4] Khayri A. M. Ali and Roland Karlsson. Full Prolog and Scheduling Or-parallelism in Muse. *International Journal of Parallel Programming*, 19(6):445–475, 1991.
- [5] H. Alshawhi and D. B.. Moran. The Delphi Model and some Preliminary Experiments. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1578–1589, August 1988.
- [6] Thomas Anderson, Brian Bershad, Edward Lazowska, and Henry Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *ACM Transactions on Computer Systems*, pages 53–79, February 1992.
- [7] Johan Andersson, Stefan Andersson, Kent Boortz, Mats Carlsson, Hans Nilsson, Thomas Sjoland, and Johan Widén. SICStus Prolog User's Manual – T93-01. Technical report, Swedish Institute of Computer Science (SICS), November 1997.
- [8] G.R. Andrews. The distributed programming language SR — Mechanisms, designs and implementation. In *Software – Practice and experience 12*, pages 719–754, August 1982.

- [9] J. Backus. Can Programming be liberated from the Von Neumann Style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613-641. ACM Turing Award Lecture., 1978.
- [10] Reem Bahgat and Steve Gregory. Pandora: Non-deterministic Parallel Logic Programming. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 471–486. MIT Press, June 1989.
- [11] J. Barklund. *Parallel Unification*. PhD Thesis, Uppsala University, 1990.
- [12] Uri Baron, Jacques Chassin de Kergommeux, Max Hailperin, Michael Ratcliffe, Philippe Robert, Jean-Claude Syre, and Harald Westphal. The Parallel ECRC Prolog System PEPsSys: an Overview and Evaluation Results. In *International Conference on Fifth Generation Computer Systems 1988*, pages 841–850. ICOT, Tokyo, Japan, November 1988.
- [13] Anthony Beaumont, S. Muthu Raman, and Péter Szeredi. Scheduling or-parallelism in Aurora with the Bristol scheduler. Technical Report TR-90-04, University of Bristol, Computer Science Department, March 1990.
- [14] Anthony Beaumont, S Muthu Raman, Péter Szeredi, and David H. D. Warren. Flexible Scheduling of OR-Parallelism in Aurora: The Bristol Scheduler. In *PARLE91: Conference on Parallel Architectures and Languages Europe*, volume 2, pages 403–420. Springer Verlag, June 1991.
- [15] Tony Beaumont and David H. D. Warren. Scheduling speculative work in or-parallel Prolog systems. In *International Conference of Logic Programming*, pages 135–149. MIT Press, 1993.
- [16] Johan Bevenmyr. *A Recursion Parallel Prolog Engine*. Thesis for the Degree of Licentiate of Philosophy, Uppsala University, 1993.
- [17] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall International Series in Computer Science, 1988.
- [18] James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfeld, Ewing Lusk, Ross Overbeek, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart, and Winston, 1987.
- [19] Per Brand. Wavefront Scheduling. Internal Report, Gigalips Project, 1988.

- [20] Burkholder. The halting problem. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 18, 1987.
- [21] A. Burns. *Programming in Occam-2*. Wokingham: Addison-Wesley, 1988.
- [22] C. Amza et al. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 19(2):18–28, February 1996.
- [23] Alan Calderwood and Péter Szeredi. Scheduling or-parallelism in Aurora – the Manchester scheduler. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 419–435. MIT Press, June 1989.
- [24] Mats Carlsson. On the efficiency of optimised shallow backtracking in Compiled Prolog. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 3–15. MIT Press, June 1989.
- [25] Mats Carlsson. *Design and Implementation of an OR-Parallel Prolog Engine*. SICS Dissertation Series 02, The Royal Institute of Technology, 1990.
- [26] Mats Carlsson and Péter Szeredi. The Aurora abstract machine and its emulator. SICS Research Report R90005, Swedish Institute of Computer Science, 1990.
- [27] L. Castro, V. Costa, C. Geyer, F. Silva, P. Vargas, and Manuel E. Correia. Daos – Scalable And-Or Parallelism. In *EuroPar 1999 Parallel Processing (EuroPar 1999)*, pages 899–908. Springer-Verlag, 1999.
- [28] J. Chassin de Kergommeaux and P. Robert. An Abstract Machine to Implement Or-And Parallel Prolog Efficiently. *The Journal of Logic Programming*, 8(3), May 1990.
- [29] Takashi Chikayama, Tetsuro Fujise, and Daigo Sekit. A Portable and Efficient Implementation of KL1. In *6th International Symposium PLILP*, pages 25–39, 1994.
- [30] Keith L. Clark and S. Gregory. PARLOG: Parallel Programming in Logic. *ACM TOPLAS*, 8:1–49, January 1986.
- [31] Keith L. Clark, F. G. McCabe, and S. Gregory. IC-PROLOG – language features. In K. L. Clark and S. Å. Tärnlund, editors, *Logic Programming*, pages 253–266. Academic Press, London, 1982.

- [32] Alain Colmerauer. *Prolog II: Reference Manual and Theoretical Model*. Groupe D'Intelligence Artificielle, Faculte Des Sciences De Luminy, Marseilles, October 1982.
- [33] Alain Colmerauer. An Introduction to Prolog-III. *Communications ACM*, 33(7):69–90, July 1990.
- [34] Alain Colmerauer and Philippe Roussekk. The birth of prolog. *ACM SIGPLAN Notices*, 28(3):37–52, March 1993.
- [35] John S. Conery. *Parallel Execution of Logic Programs*. Kluwer Academic Publishers, Norwell, Ma 02061, 1987.
- [36] Manuel E. Correia and V. Santos Costa. Iap for dummies; the yap design. In *Workshop on Parallelism and Implementation Technology for (Constraint) Logic Languages, jointly with ICLP-1999*, New Mexico, USA, 1999.
- [37] Manuel E. Correia, V. Santos Costa, and Fernando Silva. Using Sparse Binding Arrays for Or-parallelism (Extended Abstract). In *3rd COMPULOG NET Workshop on Parallelism and Implementation Technologies, Utrecht, Holland*, 1995.
- [38] Manuel E. Correia, Fernando Silva, and V. Santos Costa. Aurora vs. Muse: A Performance Study of Two Or-Parallel Prolog Systems. *Computing Systems Engineering Journal, Pergamon*, 6(4/5):345–349, 1995.
- [39] Manuel E. Correia, Fernando Silva, and V. Santos Costa. The SBA: Exploiting orthogonality in AND-OR parallel systems. In Jan Małuszyński, editor, *Proceedings of the International Symposium on Logic Programming (ILPS-97)*, pages 117–132, Cambridge, October 13–16 1997. MIT Press.
- [40] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(1, 2, 3 and 4):103–179, 1992.
- [41] J. A. Crammond. *Implementation of Committed Choice Logic Languages on Shared Memory Multiprocessors*. PhD thesis, Heriot-Watt University, Edinburgh, May 1988. Research Report PAR 88/4, Dept. of Computing, Imperial College, London.
- [42] L. Damas, V. Santos Costa, R Reis, and R. Azevedo. *YAP User's Guide and Reference Manual*, 1989.

- [43] Saumya K. Debray, Pedro López-García, Manuel V. Hermenegildo, and Nai-Wei Lin. Lower bound cost estimation for logic programs. In *Logic Programming: Proceedings of the 1997 International Logic Programming Symposium*. MIT Press, October 1997.
- [44] Doug DeGroot. Restricted and-parallelism. In Hideo Aiso, editor, *International Conference on Fifth Generation Computer Systems 1984*, pages 471–478. Institute for New Generation Computing, Tokyo, 1984.
- [45] P. Deransart, A. Ed-Dbali, L. Cervoni, and A. A. Ed-Ball. *Prolog, The Standard Reference Manual*. Springer Verlag, 1996.
- [46] Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *International Conference on Fifth Generation Computer Systems 1988*, pages 693–702. ICOT, Tokyo, Japan, November 1988.
- [47] I. Dutra. A Flexible Scheduler for the Andorra-I System. In *LNCS 569, ICLP’91 Pre-Conference Workshop on Parallel Execution of Logic Programs*, pages 70–82. Springer-Verlag, June 1991.
- [48] I. Dutra. Strategies for Scheduling And- and Or-Work in Parallel Logic Programming Systems. In *Logic Programming: Proceedings of the 1994 International Logic Programming Symposium*, pages 289–304. MIT Press, 1994.
- [49] D. Eager, E. Lazowska, and J. Zahorjan. Adaptative load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, May 1986.
- [50] Hesham El-Rewini and Ted G. Lewis. *Distributed and Parallel Computing*. Manning – Prentice Hall, 1998.
- [51] Enrico Pontelli. *Efficient Parallel Execution of PROLOG Programs*. PhD thesis, New Mexico State University, 1997.
- [52] Abderrahamane Aggoun et. all. *ECLiPSe 3.5 User Manual*. ECRC, December 1995.
- [53] Ian Foster and Steven Tuecke. Parallel Programming with PCN (Program Composition Notation). Technical Report, Argonne National Laboratory, January 1993.

- [54] Pedro López García and Manuel Hermenegildo. A Technique for Dynamic Term Size Computation via Program Transformation. Research Report, Facultad de Informática, Universidad Politécnica de Madrid, TR CLIP 8/93.1(94), March 1994.
- [55] Steve Gregory. *Parallel Logic Programming in PARLOG*. Addison–Wesley, 1987.
- [56] E. Lusk Gropp W. and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Cambridge MA: MIT Press, 1994.
- [57] G. Gupta. Paged Binding Array: Environment Representation for And-Or Parallel Prolog. Technical Report TR-91-24, University of Bristol, Computer Science Department, October 1991.
- [58] G. Gupta. *Multiprocessor Execution of Logic Programs*. Kluwer Academic Press, 1994.
- [59] G. Gupta, M. Hermenegildo, Enrico Pontelli, and V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proc. ICLP'94*, pages 93–109. MIT Press, 1994.
- [60] G. Gupta, M. Hermenegildo, and V. Santos Costa. And-Or Parallel Prolog: A Recomputation based Approach. *New Generation Computing*, 11(3,4):770–782, 1993.
- [61] G. Gupta and M. V. Hermenegildo. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *LNCS 569, ICLP'91 Pre-Conference Workshop on Parallel Execution of Logic Programs*, pages 146–158. Springer-Verlag, June 1991.
- [62] G. Gupta and M. V. Hermenegildo. Recomputation based Implementations of And-Or Parallel Prolog. In *International Conference on Fifth Generation Computer Systems 1992*, pages 770–792. ICOT, Tokyo, Japan, June 1992.
- [63] G. Gupta and Bharat Jayaraman. On Criteria for Or-Parallel Execution Models of Logic Programs. In *Proceedings of the North American Conference on Logic Programming*, pages 604–623. MIT Press, October 1989.
- [64] G. Gupta and V. Santos Costa. And-Or Parallelism in Full Prolog with Paged Binding Arrays. In *PARLE'92 Parallel Architectures and Languages Europe, LNCS 605*, pages 617–632. Springer-Verlag, June 1992.

- [65] G. Gupta and V. Santos Costa. Cuts and Side-Effects in And-Or Parallel Prolog. *Journal of Logic Programming*, 27(1):45–71, April 1996.
- [66] G. Gupta, V. Santos Costa, and Enrico Pontelli. Shared Paged Binding Arrays: A Universal Data-structure for Parallel Logic Programming. Proc. NSF/ICOT Workshop on Parallel Logic Programming and its Environments, CIS-94-04, University of Oregon, Mar. 1994.
- [67] M. Hanus. The integration of functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20, 1994.
- [68] Seif Haridi and Per Brand. Andorra Prolog—an integration of Prolog and committed choice languages. In *International Conference on Fifth Generation Computer Systems 1988*. ICOT, 1988.
- [69] Bogumił Hausman. Pruning and scheduling speculative work in or-parallel Prolog. In *PARLE 89, Conference on Parallel Architectures and Languages Europe*. Springer-Verlag, 1989.
- [70] Bogumił Hausman. *Pruning and Speculative Work in OR-Parallel PROLOG*. PhD thesis, The Royal Institute of Technology, Stockholm, 1990.
- [71] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*. PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.
- [72] M. V. Hermenegildo. An Abstract Machine for Restricted And-Parallel Execution of Logic Programs. In Ehud Shapiro, editor, *Third International Conference on Logic Programming, London*, pages 25–39. Springer-Verlag, July 1986.
- [73] M. V. Hermenegildo. Relating goal scheduling, precedence, and memory management in and-parallel execution of logic programs. In *ICLP87*, pages 556–575. MIT Press, 1987.
- [74] M. V. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

- [75] M. V. Hermenegildo and R. I. Nasr. Efficient Management of Backtracking in AND-parallelism. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 40–54. Imperial College, Springer-Verlag, July 1986.
- [76] M. V. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.
- [77] ICOT, Tokyo Japan. *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Japan, 1992. Association for Computing Machinery.
- [78] Becker D. J., Sterling T. M., Salmon J., and Savarese D. F. *Who to Build a Beowulf — A guide to the implementation and Application of PC Clusters*. Scientific and Engineering Computation Series — MIT Press, 1999.
- [79] Joxan Jaffar and Spiro Michaylov. Methodology and implementation of a CLP system. In Jean-Louis Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, pages 196–218. University of Melbourne, MIT Press, May 1987.
- [80] L. V. Kalé. The REDUCE OR process model for parallel evaluation of logic programming. In Jean-Louis Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, MIT Press Series in Logic Programming, pages 616–632. University of Melbourne, "MIT Press", May 1987.
- [81] Roland Karlsson. *A High Performance OR-parallel Prolog System*. SICS Dissertation Series 07, The Royal Institute of Technology, 1991.
- [82] R. A. Kowalski. Predicate Logic as a Programming Language. In *Proceedings IFIPS*, pages 569–574, 1974.
- [83] Robert A. Kowalski. *Logic for Problem Solving*. Elsevier North-Holland Inc., 1979.
- [84] Daniel E. Lenoski and Wolf-Dietrich Weber. *Scalable Shared-Memory Multiprocessing*. Morgan Kaufmann, 1995.
- [85] K. Li. Ivy: A shared virtual memory system for parallel computing. In *Proc. International Conference On Parallel Processing*, pages 94–101, 1988.

- [86] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *Proc. 5th ACM Symposium on Principles of Distributed Computing*, pages 229–239, 1986.
- [87] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.
- [88] Ricardo Lopes and V. Santos Costa. Design and Implementation of the BEAM, an Extended Andorra Model System. Technical Report DCC-2000-2, DCC - FC and LIACC - University of Porto, 5 2000.
- [89] Ricardo Lopes, Fernando Silva, V. Santos Costa, and Salvador Abreu. The RAINBOW: Towards a Parallel Beam. In *Workshop on Parallelism and Implementation Technology for (Constraint) Logic Languages*, London, July 2000.
- [90] D. W. Loveland. *Automated theorem proving: A logical Basis*. North Holland, New York, 1978.
- [91] Ewing Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, David H. D. Warren, A. Calderwood, P. Szeredi, Seif Haridi, P. Brand, M. Carlsson, A. Ciepelewski, and B. Hausman. The Aurora or-parallel Prolog system. In *International Conference on Fifth Generation Computer Systems 1988*, pages 819–830. ICOT, Tokyo, Japan, November 1988.
- [92] Ewing Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, David H. D. Warren, A. Calderwood, P. Szeredi, Seif Haridi, P. Brand, M. Carlsson, A. Ciepelewski, and B. Hausman. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.
- [93] Kim Marriot and Peter J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998.
- [94] Hideo Masukawa, Kouichi Kumon, Akihiro Itashiki, Ken Satoh, and Yukio Sohma. "Kabu-Wake" Parallel Inference Mechanism and Its Evaluation. In *1986 Proceedings Fall Joint Computer Conference*, pages 955–962. IEEE Computer Society Press, November 1986.
- [95] J. Harold McBeth. On the reference counter method. *Communications of the ACM*, 6(9):575, September 1963.

- [96] Johan Montelius and Khayri A. M. Ali. An And/Or-Parallel Implementation of AKL. Proc. NSF/ICOT Workshop on Parallel Logic Programming and its Environments, CIS-94-04, University of Oregon, Mar. 1994.
- [97] Johan Montelius and Khayri A. M. Ali. An And/Or-Parallel Implementation of AKL. *New Generation Computing*, 14(1), 1996.
- [98] Johan Montelius and Peter Magnusson. Using SimICS to Evaluate the PennySystem. In *Logic Programming: Proceedings of the 1997 International Logic Programming Symposium*, October 1997.
- [99] K. Muthukumar and M. V. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.
- [100] Lee Naish. *Negation and Control in Prolog*. Lecture notes in Computer Science 238. Springer-Verlag, 1985.
- [101] F. Pereira. *CProlog user's Manual, version 1.5*. EdCAAD, Department of Architecture, University of Edinburgh, Scotland, 1984.
- [102] Luís Moniz Pereira, Luís Monteiro, José Cunha, and Joaquim N. Aparício. Delta Prolog: a distributed backtracking extension with events. In Ehud Shapiro, editor, *Third International Conference on Logic Programming, London*, pages 69–83. Springer-Verlag, 1986.
- [103] Enrico Pontelli and G. Gupta. On the duality between or-parallelism and and-parallelism in logic programming. *Lecture Notes in Computer Science*, 966:43, 1995.
- [104] Enrico Pontelli and G. Gupta. Dependent and-parallelism revisited. In Michael Maher, editor, *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 542–542. MIT Press, September 2–6 1996.
- [105] Enrico Pontelli, G. Gupta, and M. Hermenegildo. Implementation and performance of &ACE: An independent and-parallel system. Technical report, NMSU, 1994.
- [106] Enrico Pontelli, G. Gupta, and M. Hermenegildo. &ACE: A High-Performance Parallel Prolog System. In *International Parallel Processing Symposium*. IEEE

- Computer Society Technical Committee on Parallel Processing, IEEE Computer Society, April 1995.
- [107] Enrico Pontelli, G. Gupta, M. Hermenegildo, M. Carro, and D. Tang. Efficient Implementation of And-Parallel Logic Programming Systems. *Computer Languages*, 22(2/3), 1996.
- [108] M. Tomašević J. Protić and V. Milutinović. A survey of distributed shared memory systems. In Trevor N. Mudge and Bruce D. Shriver, editors, *Proceedings of the 28th Annual Hawaii International Conference on System Sciences. Volume 1: Architecture*, pages 74–84, Los Alamitos, CA, USA, January 1995. IEEE Computer Society Press.
- [109] J. A. Robinson. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(23):23–41, January 1965.
- [110] Ricardo Rocha, Fernando Silva, and V. Santos Costa. Yapor: an or-parallel prolog system based on environment copying. In *9th Portuguese Conference on Artificial Intelligence (EPIA '99)*., pages 178–192. Springer-Verlag, LNAI 1695, September 1999.
- [111] V. Santos Costa. COWL: Copy-On-Write for Logic Programs. In *IPPS/SPDP'99*, pages 720–727. IEEE Computer Press, May 1999.
- [112] V. Santos Costa. Optimising bytecode emulation for prolog. In *LNCS 1702, Proceedings of PPDP'99*, pages 261–267. Springer-Verlag, September 1999.
- [113] V. Santos Costa, R. Bianchini, and I. Dutra. Evaluating the Impact of Coherence Protocols on Parallel Logic Programming Systems. In *Proceedings of the 1997 EUROMICRO Workshop on Parallel and Distributed Processing*, January 1997.
- [114] V. Santos Costa, Manuel E. Correia, and Fernando Silva. Aurora and Friends on the Sun (Extended Abstract). In *2nd COMPULOG NET Workshop on Parallelism and Implementation Technologies, Madrid*., 1994.
- [115] V. Santos Costa, Manuel E. Correia, and Fernando Silva. Performance of Sparse Binding Arrays for Or-Parallelism. In *Proceedings of the VIII SBAC-PAD, Recife-Brazil*, August 1996.
- [116] V. Santos Costa, Ricardo Rocha, and F. Silva. Novel models for or-parallel logic programs: A performance analysis. In *Euro-Par 2000 Parallel Processing (Euro-*

- Par 2000*), pages 744–753. Springer-Verlag, LNCS 1900, August/September 2000.
- [117] 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 *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOPP*, pages 83–93. ACM press, April 1991. SIGPLAN Notices vol 26(7), July 1991.
 - [118] V. A. Saraswat. Partial Correctness Semantics for $\mathbf{CP}[\downarrow, |, \&, ;]$. In *Proceedings of the Foundations of Software Technology and Theoretical Computer Science Conference*, pages 347–368, December 1985.
 - [119] Ehud Shapiro. A Subset of Concurrent Prolog and Its Interpreter. In Ehud Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 27–83. MIT Press, Cambridge MA, 1987.
 - [120] Ehud Shapiro. *Concurrent Prolog: Collected Papers*. MIT Press, 1987.
 - [121] Ehud Shapiro. The family of Concurrent Logic Programming Languages. *ACM computing surveys*, 21(3):412–510, 1989.
 - [122] Kish Shen. Exploiting And-parallelism in Prolog: the Dynamic Dependent And-parallel Scheme (DDAS). In *Proceedings of the 1992 Joint International Conference and Symposium on Logic Programming*, 1992.
 - [123] Kish Shen. *Studies of AND/OR Parallelism in Prolog*. PhD thesis, University of Cambridge, 1992.
 - [124] Kish Shen. Initial Results from the Parallel Implementation of DASWAM. In Michael Maher, editor, *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*. The MIT Press, 1996.
 - [125] Kish Shen. A new Implementation scheme for combining And/Or parallelism. In *Workshop on Parallelism and Implementation Technology for (constraint) Logic Programming Languages, Port Jefferson, New York*, 1997.
 - [126] Kish Shen and Manuel V. Hermenegildo. A Simulation Study of Or- and Independent And-Parallelism. In *Logic Programming: Proceedings of the International Logic Programming Symposium*, pages 135–151. MIT Press, October 1991.

- [127] Kish Shen and Manuel V. Hermenegildo. On the virtues of spaghetti. In *ICLP'93 Postconference Workshop on Practical Implementations and Systems Experience*, 1993.
- [128] Kish Shen and Manuel V. Hermenegildo. High Level Characteristics of Or- and Independent And-parallelism in Prolog. *International Journal of Parallel Programming*, 24(5):433–478, 1996.
- [129] Kish Shen, V. Santos Costa, and Andy King. Distance: a New Metric for Controlling Granularity for Parallel Execution. *Journal of Functional and Logic Programming*, (Special Issue 1), 1999.
- [130] Fernando M. A. Silva. *An Implementation of Or-Parallel Prolog on a Distributed Shared Memory Architecture*. PhD thesis, Dept. of Computer Science, Univ. of Manchester, September 1993.
- [131] Raéd Sindaha. Branch-Level Scheduling in Aurora: The Dharma Scheduler. In *ILPS93*, pages 403–419. The MIT Press, June 1993.
- [132] I. Sommerville and R. Morrison. *Software Development with Ada*. International Computer Science Series: Addison-Wesley, 1987.
- [133] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, 1986.
- [134] T. Sterling, D. Becker, and D. Savarese. BEOWULF: A Parallel Workstation for Scientific Computation. In *Proceedings of the 1995 International Conference on Parallel Processing (ICPP)*, 1995.
- [135] V. S. Sunderam. PVM: A framework for Parallel Distributed Computing. *Concurrency: Practice & Experience*, 2(4), 1990.
- [136] Péter Szeredi. Performance analysis of the Aurora or-parallel Prolog system. In *Proceedings of the North American Conference on Logic Programming*, pages 713–732. MIT Press, October 1989.
- [137] Péter Szeredi and Mats Carlsson. The engine–scheduler interface in the aurora or–parallel prolog system. Technical Report TR-90-09, University of Bristol, Computer Science Department, April 1990.
- [138] Akikazu Takeuchi. *Parallel Logic Programming*. PhD thesis, University of Tokyo, July 1990.

- [139] Jiro Tanaka, Kazunori Ueda, Toshihiko Miyazaki, Akikazu Takeuchi, Yuji Matsumoto, and Koichi Furukawa. Guarded Horn Clauses and Experiences with Parallel Programming. In *1986 Proceedings Fall Joint Computer Conference*, pages 948–954. IEEE Computer Society Press, November 1986.
- [140] D. Tang, Enrico Pontelli, G. Gupta, and M. Carro. Last Parallel Call Optimization and Fast Backtracking in and-parallel systems. In *Workshop in Parallel and Data-Parallel Execution of Logic Programs.*, pages 93–109, 1994.
- [141] Evan Tick. Prolog Memory-Referencing Behavior. Technical Report CSL-TR-85-281, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, September 1985.
- [142] Evan Tick. *Parallel Logic Programming*. MIT Press, 1991.
- [143] Kazunori Ueda. Guarded Horn Clauses. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 140–156. MIT Press, Cambridge MA, 1987.
- [144] Kazunori Ueda and Takashi Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *Computer Journal*, December 1990.
- [145] Uresh Vahalia. *Unix Internals, the new frontiers*. Prentice Hall, 1996.
- [146] M. H. van Emdem and G. J. de Lucena Filho. Predicate Logic as a Language for Parallel Programming. In K. L. Clark and S. Å. Tärnlund, editors, *Logic Programming*, pages 189–198. Academic Press, London, 1982.
- [147] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, November 1990.
- [148] D. H. D. Warren. Implementing Prolog - Compiling Predicate Logic Programs. Technical Report 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.
- [149] D. H. D. Warren, L. M. Pereira, and F. C. N. Pereira. Prolog—The Language and its Implementation Compared with Lisp. *ACM SIGPLAN Notices*, 12(8):109–115, 1977.
- [150] David H. D. Warren. *Applied Logic—Its Use and Implementation as a Programming Tool*. PhD thesis, Edinburgh University, 1977. Available as Technical Note 290, SRI International.

- [151] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.
- [152] David H. D. Warren. The SRI model for or-parallel execution of Prolog—abstract design and implementation issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, 1987.
- [153] David H. D. Warren. Extended Andorra model. PEPMA Project workshop, University of Bristol, October 1989.
- [154] Harald Westphal, Philippe Robert, Jacques Chassin de Kergommeaux, and Jean-Claude Syre. The PEPSys model: combining backtracking, and- and or-parallelism. In *The 1987 Symposium on Logic Programming, San Francisco, California*. IEEE, 1987.
- [155] R. Yang, T. Beaumont, I. Dutra, V. Santos Costa, and D. H. D. Warren. Performance of the Compiler-based Andorra-I System. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 150–166. MIT Press, June 1993.
- [156] Rong Yang. *P-Prolog a Parallel Logic Programming Language*. World Scientific, 1987.
- [157] Rong Yang and Hideo Aiso. P-Prolog: a Parallel Logic Language Based on Exclusive Relation. In Ehud Shapiro, editor, *Third International Conference on Logic Programming, London*, pages 255–269. Springer-Verlag, July 1986.
- [158] Rong Yang and V. Santos Costa. Andorra-I: A system integrating or-parallelism with dependent and-parallelism. Technical Report TR-90-03, University of Bristol, Computer Science Department, March 1990.
- [159] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1991.