High Performance Parallel Logic Programming on Distributed Shared Memory Multiprocessors

Hiecheol Kim

CENG 96-19

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213) 740-4484

September 1996

HIGH PERFORMANCE PARALLEL LOGIC PROGRAMMING ON DISTRIBUTED SHARED MEMORY MULTIPROCESSORS

by

Hiecheol Kim

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Computer Engineering)

AUGUST 1996

UNIVERSITY OF SOUTHERN CALIFORNIA
THE GRADUATE SCHOOL
UNIVERSITY PARK
LOS ANGELES, CALIFORNIA 90007

This dissertation, written by

HIECHEOL KIM

under the direction of his *Dissertation Committee, and approved by all its members, has been presented to and accepted by The Graduate School, in partial fulfillment of requirements for the degree of*

DOCTOR OF PHILOSOPHY

[illegible]
Dean of Graduate Studies

Date Aug 20 1996

DISSERTATION COMMITTEE

[illegible]
Chairperson

[illegible]

[illegible]

Dedication

To my parents Won-suk Kim and Sun-won Lee.

Acknowledgements

I am deeply grateful to my advisor, professor Jean-Luc Gaudiot, for his guidance, support, and encouragement throughout my graduate studies at University of Southern California. His thorough scientific approach and unending quest for excellence have been inspirational during the years of my thesis research.

I would like to thank professor Alvin Despain and professor Doug Ierardi for serving on my dissertation committee. I sincerely appreciate the time and guidance they provided in the completion of my dissertation. I would also like to thank professor Ellis Horowitz, Michel Dubois, and professor Massoud Pedram for their valuable comments and discussions.

Many thanks are due to my colleagues in the USC PDPC (parallel distributed processing center). Dr. Namhoon Yoo, Dr. Dae-kyun Yoon, Wenyen Lin, Eric Tsen, Chung-ta Chung, and Chulho Shin have closely worked with many insightful discussions and comments. Also, my former colleagues , Professor Chin-hyun Kim and Professor Andrew Shon encourage me a lot when I began my dissertation. Also, I really appreciate Neung-soo Park, You-Pyo Hong, and Yung-ho Choi for their help. Special thanks goes to Mary Zittercob, Rohini Montenegro, and Joanna Wingert for their assistance.

Several organizations and individuals made this dissertation possible with many insightful discussions and equipment support. Dr. Cristine Montgomery and Robert E. Stumberger of Language Systems Inc., and Dr. Raymond Liuzzi in Rome Laboratory closely collaborated with me during the whole period of this thesis. Also, I would like to acknowledge the support by the Air force Systems Command, Rome Laboratory/C3CA, Griffiss Air force Base, NY, under Contract No. F30602-91-C-0130. Fujitsu Laboratories, Ltd. in Japan permits me to use the AP1000 parallel machine for my experimentation.

Many thanks are due to my friends Kangwoo Lee, Edward Kim in TRW, Suk-hee Kim, Junho Ha, Donghyun Heo, Jong-Bum Lee and numerous others including Dr. Sung-bok Kim, and Dr. Sungwook Kim for their friendship and sharing their valuable time in many occasions. I would also like to express special thank to professor Yong-Doo Lee at Taegu university and professor Sin-Duck Kim in Yousei university for their encouragement and support.

I sincerely thank my parent for his care and love. I also thank many saints in Cana Presbyterian church, particularly Reverend Sim, Puyng-hwa Kim's, and Sin-mo Lee's family for their fellowship in Christ. Most of all, I really thank God who has always been with me, showing His unfailing love.

Contents

List Of Tables

List Of Figures

Abstract

Logic programming has drawn growing attention as an important programming paradigm due to its high programmability and implicit parallelism. Can logic programs achieve high performance on general purpose large-scale parallel architectures? The answer consists in how to efficiently exploit parallelism in logic programs subject to the architectural features of such systems.

Parallel execution models for logic programs developed so far suffer from the lack of flexibility with regard to the optimizations which are essential in the stage of task scheduling. In other words, some execution models put some restriction on the types of scheduling algorithms which can be implemented in their runtime scheduler. Besides, some parallel execution models are not efficient with regard to such optimizations. In other words, even if a wide range of optimizations can be implemented in the runtime scheduler, the lack of efficiency prevents us from obtaining expected performance gain from such optimizations. Therefore, when pursuing performance logic programming on large scale multiprocessors, we ought to clearly address the issues of flexibility and efficiency in the design of the parallel execution model.

In this dissertation, a new parallel execution model for logic programs and its performance are presented. Our execution model is designed particularly with focus on high flexibility and efficiency of optimizations with regard to architectural features and scheduling algorithms. It is based on the findings obtained from an analysis of ideal OR-parallel systems. Theoretically, an OR-parallel system is considered *ideal* if it satisfies the constant time condition for all the following three performance criteria: variable access, task creation, and task switching. It has been known that the exploitation of OR-parallelism is fundamentally limited because no ideal OR-parallel systems exist in the entire design space. In our analysis, we, however, proved that ideal OR-parallel logic programming systems exist in the theoretic design space. The result also reveals that an ideal OR-parallel system is possible *only* when some semantic information is used in the representation and management of conditional variables.

Our execution model utilizes two kinds of semantic information: (1) the inference depth and (2) the least common ancestor relation between nodes in the runtime search tree. The inference depth of a node refers to the number of choice points allocated prior to the execution of the node. The utilization of such semantic information enables the execution model to perform the scheduling with respect to a common ancestor tree rather than an ordinary search tree. A common ancestor tree is a representation of a runtime program execution based on the common ancestor node relation. It is quite simpler than the ordinary search tree, consisting only of the least common ancestor nodes between those nodes on which processors are standing. As a result, the scheduler can carry out various scheduling activities with utmost efficiency.

Based on our execution model, we designed a parallel abstract machine for Prolog and subsequently implemented the abstract machine and its runtime system in a HP's SPP IAX-0016 distributed shared memory multiprocessor. The implementation also incudes a front-end compiler, that produces abstract machine code from a Prolog program, and a translator which generates from the abstract machine code a parallel-C code executable on a SPP-IAX system. Moreover, to show that a wide range of scheduling algorithms can be efficiently supported in our execution model, we implemented both a top-most scheduling and a bottom-most scheduling in the runtime scheduler. Furthermore, to show the efficiency of optimizations tailored particularly toward architectural specifications, we implemented an architectural optimization to reduce the amount of remote memory accesses within a top-most tree-based scheduling strategy. The performance result and its analysis show that our execution model is highly flexible as well as efficient for both algorithmic and architectural optimizations which are carried out by runtime schedulers. Furthermore, the results clearly validate the hypothesis that the architectural and algorithmic optimizations are crucial for expected performance on large-scale parallel machines.

Chapter 1

Introduction

This chapter presents the motivation, goals, and the contributions of the dissertation.

1.1 Motivation

Logic programming has many advantages as a parallel programming paradigm since it is very easy to program and retains high expressive power due to its declarative semantics. Indeed, Prolog, the most popular logic programming language, is accepted as one of the most important and widely used computer programming languages and we have seen increasing use of Prolog for a wide range of applications in symbolic computing, particularly in natural language and knowledge-based processing. As logic programs become more realistic and they frequently require large amount of computation, parallel logic programming has been recognized as a promising way to improve the performance of logic programs. Indeed, many researchers have pursued the goal of designing efficient techniques for parallel logic programming systems.

Recently, we have seen a great opportunity for high performance parallel logic programming. On the parallelism side, logic programs present many types of parallelism, *e.g.*, OR-, dependent AND-, independent AND-, stream AND-, and unification-parallelism [10]. On the architecture side, a number of large-scale parallel architectures become realistic for the users. Indeed, some distributed memory architectures are commercially available in which the number of processing elements (PEs) is over an order of hundreds. Researches to solve some fundamental problems, that limit the number of PEs in a shared memory system, have also yielded promising result for scalable shared-memory systems. Also, as workstations have spread very rapidly and the trend is predicted to continue, clusters of workstations also become a promising scalable parallel platform.

However, to get the expected speedup of logic programs on large-scale parallel architectures is very hard. It is because the internal inference mechanism of logic languages and their execution behaviors are quite different from the conventional languages, as explained below:

- The internal inference mechanism of logic languages, more precisely the principle of SLD refutation [61], requires the maintenance and traversal of a search tree at runtime. The dynamic nature of the search tree makes program partitioning and allocation very hard and demands highly efficient runtime schedulers.
- The sequential nature of logic languages complicates their parallel execution. For example, the selection rule, *e.g.,* selection from left to right in PROLOG, entails stricter control of parallelism.
- Logic programs, executed in parallel, show speculative behaviors for some types of parallelism. The size of the search space depends on the parallel search strategy used, which, in turn, influences the total execution time. Search strategies efficient enough in the presence of such speculative aspects are crucial for the minimization of the search space.

In spite of such difficulties, a number of *parallel execution models* (PEMs) have been explored for logic languages. Such models are mostly geared toward uniform memory access (UMA) small-scale shared memory multiprocessors. Among them, the PPED [17], the Muse [5], the PEPSys [54], and the Aurora [15] have been quite successful; they maintain high single thread performance, exploit a wide range of parallelism of different kinds, and achieve reasonable speedup. However, within a large-scale parallel environment, these execution models may not achieve the same success due to the following reasons:

- In most execution models, scheduling cost is very high. The real world applications tend to exhibit search trees in quite irregular shapes and the amount of scheduling activities grows rapidly as the size of system becomes larger. In such situations, high scheduling cost becomes a barrier against high performance.
- In some execution models [5], runtime scheduling strategies are tightly coupled with the execution models such that only some specific types of scheduling strategies are allowed. In other words, the algorithmic optimizations which can be implemented in runtime scheduling are limited. For example, in the Muse [5], the implementation of top-most scheduling strategies is not practically possible. However, in large scale parallel logic programming system, scheduling heuristics which employ several different scheduling algorithms are advantageous. In such systems, a scheduler can accomplish high performance by choosing an appropriate scheduling algorithm. For example, when a system becomes abundant with fine grain tasks, a top-most scheduling is rather advantageous than a bottom-most scheduling. On the other hand, for logic programs retaining large amount of speculative work, a bottom-most scheduling is more advantageous than a top-most one.
- Some models are limited in architectural optimizations such as the minimization of the total amount of the remote memory accesses. It is because such models fail to provide efficient support for these optimizations and thus suffer from unacceptable overhead in the course of the optimizations.

As another line of research to achieve scalable parallel implementations of logic languages, several models have been developed toward distributed memory multiprocessors: the OM [51], the ROMP [52], the 3DPAM model [49]. The ROMP model is practically implemented on a number of distributed memory multiprocessors and shows very promising results [52]. However, these models have some critical limitation in the realization of the expected scale-up in performance due to the following reasons:

- The single thread performance is very low because the binding environments designed for distributed memories incur intolerable overhead from side effects such as structure copying and closing operations [24].

- Distributed implementations rely on process based execution methods (execution based on procedural interpretation) rather than thread based parallel execution methods (execution by multiple sequential engines). In such implementations, the grain size cannot be flexibly controlled due to the execution behavior resulting from procedural interpretation, which, *we believe*, will cause severe limitations to the design of an efficient scheduler.
- Compared with OR-parallelism, the exploitation of AND-parallelism is extremely inefficient. It is because AND-parallelism is antithetic to OR-parallelism in that the environments referred to and created by *AND*-*parallel* tasks are not independent.

From the above observation, we believe that distributed memory multiprocessors with a non-single address space are effective merely for a small subset of logic programs that show shallow and regular search trees as well as retain very large amount OR-parallelism. In consequence, they would not be a prime choice as a platform for large-scale parallel logic programming.

Accessed crudely, the previous parallel models have no restrictions but did not clearly address the efficiency issues pertaining to their implementation on large-scale multiprocessors, as summarized below: (1) As the size of system becomes larger, the scheduling cost becomes higher and thus the expected speedup becomes harder to obtain. (2) The study based on the implementation on true large-scale multiprocessors is not available yet, even though the verification of performance through practical implementation is essential. In these respects, many challenges still exist for making parallel logic programming on large-scale parallel machines realistic.

The rationale of the dissertation

When pursuing a large-scale parallel logic programming system, we ought to consider a number of points. Especially, the following two points are noteworthy. (1) Large-scale parallel machines present several architectural characteristics that are not exhibited in small-scale parallel machines. As an example, the latency of memory accesses becomes larger as more processors are added to the system. (2) Logic programs, executed on large-scale parallel machines, present a number of execution behaviors which are not shown on small-scale machines. For example, scheduling activities such as task switching increase rapidly as the size of system becomes larger [41]. These points lead us to make the following assertions. To get any useful large-scale parallel implementation of logic programming, our prime concern must consist in how to efficiently integrate the new architectural characteristics of large-scale parallel machines with parallel logic programming techniques. Moreover, our attention must be on the alert for how to efficiently manage the behaviors of logic programs which appear newly on large-scale multiprocessors such as the rapid increase of scheduling activities.

To accomplish the efficient integration of architectural features, a set of optimizations , (which we will call *architectural optimizations* since they concern with the architectural specifications), are mandatory. As pointed out earlier, in large-scale multiprocessors, the latency of memory accesses usually changes depending on the physical distance between processors. Indeed, for these systems, the architectural optimization which will reduce the total amount of remote accesses is crucial for the expected system performance. On the implementation side, these architectural optimizations are mostly within the scope of runtime scheduling. For example, to reduce the total amount of remote access, the task selection must be carefully made by the runtime scheduler. In this sense, the scheduler must be capable of implementing such optimizations.

To achieve the efficient management of program behaviors, a set of optimizations, (which we will call *algorithmic optimizations* since they are closely related scheduling algorithms), are essential. For example, depending on the nature of applications in terms of the amount of parallelism and speculative works, some specific scheduling algorithm would be suitable for higher performance over the others. Algorithmic optimization to choose the best one is essential for hinger performance. Viewed from the perspective of the implementation, these algorithmic optimizations are inherently the subject of runtime scheduling. In this respect, the scheduler must be capable of implementing such optimizations as well.

Given a parallel execution model for a logic language, scheduling parallel tasks relies heavily on a number of operational aspects presented by the parallel execution model. Only when the implementation issues of task scheduling are efficiently addressed by the execution model, the task scheduler can contribute to higher system performance by achieving efficient implementation of them. In this sense, parallel execution models for large-scale parallel machines must be highly flexible and efficient with respect to both architectural and algorithmic *optimizations*. Here, for an execution model to be flexible to an optimization means that the execution model does not restrict the implementation of the optimization by the runtime scheduler. Also, to be efficient to an optimization means that the execution model provides enough parallel support such that the system can benefit from the implementation of the optimization.

In the exploration of our parallel execution model, we will primarily consider OR-parallelism. OR-parallelism in logic programs has been one of the most important source of parallelism in the design of parallel logic programming systems due to its relatively large granularity and simplicity of management. On large-scale parallel machines, a parallel logic programming system needs to exploit a combination of OR- and AND-parallelism to get sufficient amount of parallelism. In such a system, the efficiency in exploiting AND-parallelism would be quite tightly related to the way that OR-parallelism is managed. The reason is that the binding environment employed in support of OR-parallelism usually affects directly or sometimes indirectly the way that AND-parallelism will be managed. This dissertation considers primarily OR-parallelism due to the limitation of available resources in terms of implementation efforts, while AND-parallelism will be left as a future research. However, we believe that more robust OR-parallel logic systems lead us to envision more efficient AND/OR parallel logic programming implementations.

As an architectural platform for large-scale parallel logic programming, those with a single address space are more viable than multiple address spaces, as discussed in the previous section. In this dissertation, distributed shared memory multiprocessors have thus been chosen as the target experimental platform. The main reason for this choice is that such architectures are readily available and exhibit most inherent features of large-scale parallel machines such as the longer latency of remote memory accesses. Besides, the stable programming environments provided by those systems help us to build a robust experimental prototype.

1.2 Goals

In the previous section, we see that in order to fully materialize the expected scale-up in performance on large-scale parallel architectures, much effort is still needed both at the high level (model of execution, scheduling, etc.) and at the low level (binding environment, etc.). The primary goal of this dissertation is thus to explore a parallel execution model which is highly flexible and efficient with respect to architectural and algorithmic optimizations, while ensuring the following two points:

1. high single thread performance, and
2. low scheduling cost to adapt highly irregular shapes of the search tree on real world application programs.

Besides, this dissertation will address the performance issues based on the practical implementation of the proposed execution model.

1.3 Contribution

The research performed in this dissertation is briefly summarized as follows. To begin with, a comprehensive analysis of OR-parallel systems has been conducted within the context of large-scale parallel logic programming. As a main result of the analysis, it has been proved that ideal OR-parallel logic programming implementations are theoretically possible in systems with finite numbers of processors. The analysis has shown that an ideal OR-parallel system is possible *only* when some semantic information is used in the representation and management of conditional variables as well as in runtime scheduling. Based on the result, a parallel execution model has been designed and its performance has been evaluated.

This dissertation contributes to identifying the complexity, exploring a methodology, and evaluating the performance of a parallel logic programming system on large-scale parallel architectures, particularly on distributed shared memory multiprocessors. It also contributes to addressing various issues in relevant research disciplines which include compiler construction, parallel execution models, and runtime scheduling. These contributions are illustrated as follows.

Analysis of the ideal OR-parallel system

In the analysis, we provide a definition that allows us to classify the binding environments into two groups: static and dynamic binding environments. For each class, we have developed an analytic framework that helps us analyze the performance of OR-parallel systems. The framework consists of a set of definitions for *attributes* which illustrate the organization of a binding environment. It also has definitions for *generic properties* which illustrate the operational characteristics of a binding environment. Given the attributes, each combination corresponds to one design of a binding environment; all the set of the combinations represent the entire design space of binding environments. Based on the framework, a comprehensive analysis has been carried out respectively of static and of dynamic binding environments.

The analysis of static binding environments consists of the following two steps. First, for each generic property, we have derived all the conditions, which guarantee the property. Each condition is represented by some combinations of particular attributes. Second, we have derived the constant time conditions for each performance criterion, in which each condition is represented by a set of combinations of generic properties. The above two steps allow us to represent the constant time conditions for each performance criterion by some combinations of attributes. From these steps, it is possible to identify the performance criteria for each binding environment in the entire design space. The analysis results show that no design of *static* binding environments exist which guarantees an ideal OR-parallel system. Moreover, we have identified the constant time characteristics of the performance criteria for some existing binding environments by applying the analysis result.

The analysis of dynamic binding environments is performed differently from that of the static binding environments. In the analysis, specific to each performance criterion, the sources which result in the non-constant time operations have been identified. By using some semantic information such as inference depth we have explored some alternative implementations which avoid the non-constant time operations. As the result, it have been proved that there exist ideal OR-parallel systems under dynamic binding environments. It should be noted that ideal OR-parallel systems found in the analysis are not necessarily more efficient when they are practically implemented, than OR-parallel systems which fail to satisfy constant time conditions for some performance criterion. However, because an ideal OR-parallel system satisfies the constant time conditions for all the three performance criteria, it will provide an invaluable theoretic foundation for the design of efficient OR-parallel systems.

TCWAM: a translation based sequential Prolog implementation

The TCWAM (Thread-C code WAM) system is a Prolog compiler that translates a Prolog program into a C language program. It uses the Warren Abstract Machine (WAM) [78] as the intermediate representation for program compilation. The primary motivation of the TCWAM is to provide an experimental prototype which will be used as the sequential engine in our parallel logic programming system. The TCWAM supports the standard Prolog syntax and libraries, and also supports modules which allow modular program development.

The main feature of the system is that the execution speed is very high. It is because the TCWAM is based on the translated execution of Prolog programs instead of emulation. Compared with the other translation based approaches, it provides higher performance in terms of execution speed, generated code size and compilation speed of the generated C code. Compared with the WAMCC [29], which has been the fastest translation based Prolog system, the TCWAM is about 30 percents faster and the generated code is about 40 percent smaller.

Development of Flat indexing technique: an indexing technique for OR-parallel logic programming

To find an optimal indexing is quite complex and also demands a large number of abstract machine instructions for its implementation. In the indexing scheme of the WAM, the first argument of a clause is used as a key. Viewed from the trade-off between the efficiency and simplicity, the usage of the first argument as a key for indexing is a quite reasonable choice. However, as for OR-parallel *execution*, the indexing scheme of the WAM is inefficient in terms of parallelism. The main reason is that an invocation of some goal results in the creation of more than one choice points. In this research, we have investigated the problem in detail and have designed a new indexing scheme, which we will call *flat indexing*. Under the flat indexing scheme, the maximum number of choice points created for an invocation of a goal is always one. This makes it possible to expose parallelism earlier than the indexing scheme of the WAM. As a result, the degree of parallelism of each node becomes higher. The evaluation results show that one half of benchmarks benefit from the flat indexing and the total number of choice points created for a Prolog program is reduced by about 15 percents. As a side effect, the execution speed of the TCWAM improves by about 22 percents.

Design of a parallel execution model

In this research, we have designed a parallel execution model of Prolog, particularly suitable for large-scale parallel logic programming. In the execution model, inference depth is used for the representation of bindings made for conditional variables. Based on the representation, a new binding environment, which we will call *Tagged Binding Environment* (TBA), has been developed. The TBA is aimed at minimizing the overhead of task switching, while the cost of variable accesses is kept low. It can thus be efficiently adapted to the situation in which the amount of scheduling is very large. In the execution model, the search tree is represented in a simple form, which we will call a *common ancestor tree*. The new representation provides an opportunity of high efficiency for activities made by a runtime scheduler. The execution model provides a parallel support which enables a runtime scheduler to manage parallelism efficiently with respect to the common ancestor tree. The execution model helps a runtime scheduler to efficiently locate available works in the search tree. Moreover, it allows a runtime scheduler to efficiently identify the relative positions between processors with respect to the search tree. This enables runtime schedulers to carry out a variety of optimizations renders their implementations efficient.

Prototype OR-parallel implementation and performance evaluation

In order to verify the concept and performance of the proposed execution model, we have built an experimental prototype on a HP's SPP 1XA-0016 distributed shared memory parallel machine. The prototype implements the execution model designed in the previous research and employs the TCWAM as its sequential engine as well as the flat indexing technique. The main components of the prototype are a parallel version of the TCWAM compiler and a runtime scheduler which implements several kinds of scheduling strategies. The prototype is so far the first practical implementation of parallel Prolog on a distributed shared memory multiprocessor.

The performance evaluation has been conducted in an effort to identify the performance, to verify the underlying hypothesis, and to validate the potential of the execution model. The performance has been identified with the benchmarks which have been widely used in other systems [7, 15] because their parallelism and execution behaviors have been clearly understood. The underlying hypothesis has been verified through the comparison and analysis of the performance obtained respectively with three different scheduling strategies. The potential of the execution model has been validated by comparing the performance with other systems.

1.4 Overview of the Thesis

The rest of this thesis is organized as follows. Chapter 2 offers a review of logic programming paradigms, Prolog, and parallel execution models of logic programs. Chapter 3 presents out analysis result of OR-parallel logic programming systems and its application to existing systems. Chapter 4 presents our analysis result of ideal OR-parallel logic programming systems. Chapter 5 presents the parallel execution model proposed in this dissertation for distributed shared memory multiprocessors. Chapter 6 presents the TCWAM, developed as the sequential Prolog engine of our parallel logic programming system with focus on its new Prolog to C translation technique and sequential performance. Chapter 7 presents our analysis of indexing schemes conducted with a view to identify the relation between indexing schemes and OR-parallelism and illustrates our flat indexing technique developed to enhance the exposition of OR-parallelism. Chapter 8 presents the design, implementation, and evaluation of our OR-parallel logic programming system of Prolog implemented on a HP's SPP IXA-0016. Chapter 9 concludes the dissertation and offers future research issues.

Chapter 2

Background

This chapter presents a review of logic programming which includes an introduction to several prominent logic programming paradigms, a description of the syntax and operational semantics of Prolog, and parallel execution models of logic languages and their implementations.

2.1 Logic Programming

Logic programming, in general, is an attempt to implement Colmerauer and Kowalski's idea that logic can be used as a programming language [57]. The key motivation for logic programming is to separate the specification of *what* the program should do from *how* it should be done. Kowalski formulates this in an equation: Algorithm = Logic + Control [57]. This section presents a set of prominent logic programming paradigms.

2.1.1 Concurrent logic programming

One of the major barriers to the implementation on parallel machines of logic programming is the "binding of variables" which cause problems such as "consistency" and "multiple environment handling". In order to efficiently implement logic languages by overcoming the barriers, concurrent logic languages nondeterministically select one alternative clause and discard the others. They thus rely on the exploitation of stream-AND parallelism.

Originally, concurrent logic languages resulting from the research in *Relation Languages* [20]. They are featured with shared logical variables and *committed choice nondeterminism* (or *don't care nondeterminism*).

In general, a concurrent logic program is a finite set of guarded Horn clauses of the following form:

$$H \leftarrow G_1, \ldots, G_m \mid B_1, \ldots, B_n. \qquad m \geq 0, n \geq 0.$$

where H and B_i's are atomic formulas defined in logic languages in general and the predicates of G_i's are in fixed set $\mathbf{T}$ of guard predicates provided with concurrent logic languages. Declaratively, the *commit* operator "|" is also a conjunction operator.

Under the operational semantics, a machine state consists of a resolvent of processes and a compiled program. The reduction process then chooses one of these guards nondeterministically and *commits* to its clause. It aborts execution of the other guards and executes the body of the selected clause. State transition occurs upon the reduction of a process to the processes corresponding to goals in the body of the clause which is committed among candidate clauses with successful unification and guard evaluation. The computation is one of three states: *success*, *fail*, and *deadlock* as a result of state transitions.

The semantics of the languages are somewhat concerned with implementation details such as synchronization of shared variables (as required for stream manipulation), multiple environments and a hierarchical computation structure during execution. While concurrent logic languages in general contain many good opportunities for parallelism, the semantics of the languages depend on the operational behavior of the programs. In spite of easy implementation and parallelism control, one major drawback of concurrent logic programming is the inability of the completeness of search.

The example concurrent logic programming languages are Concurrent Prolog [72], Flat Concurrent Prolog [72], PARLOG [21], and Guarded Horn Clauses (GHC) [75], and P-Prolog [83]. Flat Concurrent Prolog language [72] supports the semantics of read-only annotation, *i.e.*, any attempt to instantiate a read-only variable with a non-variable object suspends until the writable counter-part of the read-only variable is instantiated with a non-variable object, the standard unification is extended to read-only unification where a unification that attempts to instantiate a read-only variable suspends until the variable becomes instantiated and a read-only variable points a writable variable when a read-only variable is unified with a variable. In PARLOG, processes communicate through shared logical variables and synchronize by suspending on unbound shared variables. It provides a mechanism for specifying which processes may generate a binding for a variable such that for every relation, a *mode declaration* must be given that specifies which arguments are input and which are output. For reasons of simplicity and ease of implementation, most of the recent efforts in concurrent logic programming languages allow only "flat" subsets in the guards [72, 75].

2.1.2 Constraint logic programming

Constraint Logic Programming (CLP) is a merger of two declarative paradigms: constraint solving and logic programming. Crudely stated, constraint logic programming can be viewed as the incorporation of constraints and the constraint "solving" method in a logic language. Work on CLP has mostly been devoted to languages based on Horn clauses. CLP languages are featured with the domain of constraints. In fact, Prolog can be said a CLP language where the constraints are equations over the algebra of terms which are implicit in the use of unification. Almost every CLP language employs Prolog-like terms along with other terms and constraints. We briefly introduce some of the CLP languages with their domains of constraints.

CLR(R) [47] has linear arithmetic constraints and computes over the real numbers. Nonlinear constraints are ignored until they become linear. CHIP [31] and Prolog III [22] compute over several domains: boolean, linear arithmetic and strings; Prolog III computes over the well-known 2-valued Boolean algebra and CHIP over a larger Boolean algebra that contains symbolic values. Both CHIP and Prolog III compute and perform linear arithmetic overbounded subsets of the integers (sometimes called "finite domain"). Prolog III also computes over a domain of strings. Several other languages, including clp(FD) [30], Flang [62], and cc(FD) [76], also compute over finite domains in the manner of CHIP.

LOGIN [2] and LIFE [3] compute over an order-sorted domain of feature trees. This domain provides a rather limited notion of object as in the object-oriented ways. The term syntax supported by this languages is not first-order, although every term can be interpreted through first-order constraints. Unlike other CLP languages/domains, Prolog-like trees are essentially part of this domain, rather than being built on top of the domain.

BNR-Prolog [67] computes over three domains: the 2-valued Boolean algebra, finite domains, and arithmetic over the real numbers. Trilogy [77] computes over strings, integers, and real numbers. CAL [1] computes over two domains: the real numbers, and a Boolean algebra with symbolic values. L_λ [63] and Elf [68] are derived from λ-Prolog [64] and compute over the values of closed and typed lambda expressions.

2.1.3 Prolog

As the most popular logic programming language invented by Alain Colmerauer and his associates in Marseilles France around 1970, Prolog stands for PROgramming in LOGic. Prolog implements a subset of the first order logic. Its theoretical foundation came from Horn clauses and resolution, but Prolog quickly grew to include "extra-logical" features that made it a complete programming language with more efficiency and programmability than just a pure logic theorem prover. A Prolog program provides logical specifications of what should be done (the logic), while it is executed through logical resolution (the control). This subsection provides an overview of the important syntax, semantics, common terminology, and some implementation details of Prolog. A more in-depth discussion is found in [61].

A *program* in Prolog, also called a *definite* program, is composed of a finite set of *definite clauses*, or simply *clauses* which correspond to some form of a Horn clause. Each clause contains a head followed by a body consisting of zero or more positive literals (syntactically called <goal>). A goal normally consists of a predicate symbol prefixed to argument lists surrounded by "()". Each clause is a logic sentence describing a single rule. That is, a head (consequence) is true if all body goals (antecedents) can be shown to be true. As a particular instance, if no goals exist in the body, the head is always true. A goal is true depending on the operator in the goal. A positive literal is true if the predicate can be shown to be true for some set of values bound to the variables in the atom. A negative literal is not true if the atom can be shown to be true; otherwise, the negative literal is considered to be true. A *predicate* identified by the symbol and its arity (the number of argument terms) is defined as the collection of all clauses with the same predicate symbol and arity in their head. Each predicate defines a logical relation such that a predicate is true if any clause in the predicate definition can be shown to be true.

A variable in Prolog is *logical.* That is, it can refer to an (possibly unknown) object of various types. A variable starts out uninstantiated. As more is known about the object to which a variable refers, the variable is further instantiated. For example, a variable `X` is instantiated with an object `has_book(Student,Book)`, which means that a student, referred to as `Student`, has a book, referred to as `Book`. Later, it might be determined that `Student`'s book is a text in the area of computer science published in `cs-publisher`, thereby further instantiating the value of `X` with `has_book(Student, text(cs, cs-publisher))`.

The instantiation of variables in Prolog is performed through unification that is basically a form of pattern matching. The unification in Prolog finds the *mgu* (the most general unifier) for given two terms, a goal and a clause head. Consider the following two terms:

```
Goal: f(X, g(Y,5), Y)
Head: f(3, g(4,5), Z)
```

The unification of the two terms, `Goal` and `Head`, is performed by producing a series of substitutions as below:

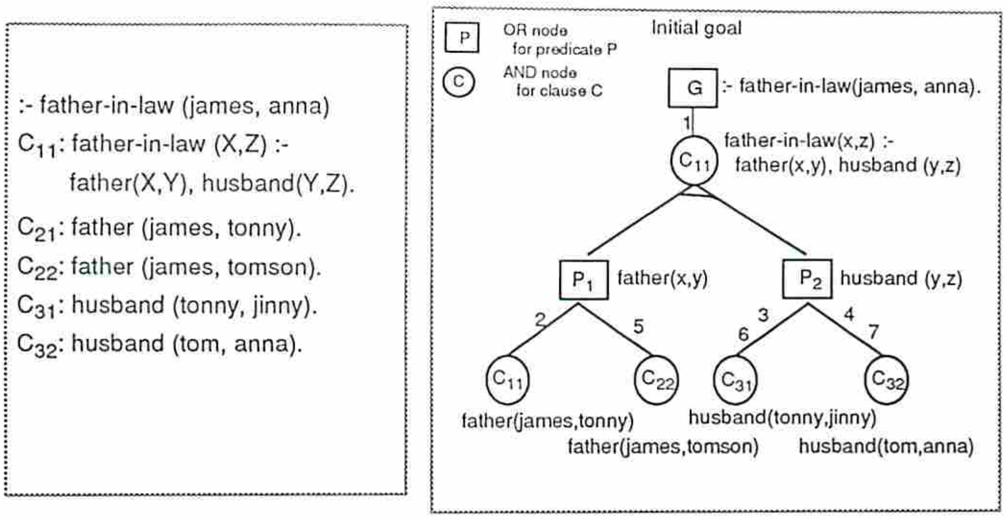
```
1 :  Unify(Goal, Head) = Unify (f(X, g(Y,5), Y), f(3, g(4,5), Z))
     => substitution = {X/3, g(Y,5)/g(4,5), Y/Z}
2 :  Unify(g(Y,5), g(4,5))
     => substitution = {Y/4}
```

After unification, `Head` and `Goal` become f(3,g(4,5),4). If a substitution cannot be produced due to some conflict in pattern matching, the unification fails. Unification may cause two or more variables to be bound to one another, *e.g.,* Y and Z in the above example. This is referred to as variable *aliasing*. They are implemented with a chain of pointers such that only a single object exists as the binding for all the aliased variables. Accesses to any of those variables thus entail traversing the pointer chain. This is referred to as *dereferencing*.

Given a query, Prolog execution begins with an attempt to prove the subgoals in the query with respect to facts and rules in the program. Proving a subgoal is done by calling the predicate whose symbol and arity match with those of the subgoal. To call a predicate, the goal literal is unified with the head of the first clause, resulting in bindings of variables in the clause. If unification fails, the clause fails, and Prolog moves on to the next clause. If all clauses fail, the subgoal fails. On the other hand, if Prolog encounters a clause with successful head unification, it attempts to prove the body goals of the clause, under the variable binding done at unification. If it fails, Prolog again moves on to the next clause, which is referred to as *backtracking*. The predicate call and backtracking strategy in Prolog result in a depth-first traversal of the Prolog execution tree.

Consider the example program in Figure 2.1 (a). The query is to prove that `father-in-law(james, anna)` is true, *i.e.*, to prove that `james` is the father-in-law of `anna`. To prove the query, Prolog first unifies the query with the head of the clause for "father-in-law/2". It then tries to show that the subgoals in "father-in-law/2", "father/2", and "husband/2" are true. Figure 2.1 (b) shows the AND/OR tree for the program, where the numbers on the arcs indicate the order in which Prolog will traverse the tree.

When backtracking, Prolog returns to the last point where the head unification was successful (the most recent *choice point*) and resumes trying other clauses. Indeed, in Figure 2.1, the failure of the unification at point 4 causes backtracking to node P_1 and to proceed to point 5. At backtracking, Prolog should undo all substitutions generated by the procedures associated with the goal that failed. For this, whenever a variable is instantiated, the variable is *trailed* by placing the name of the variable in an appropriate memory area. At backtracking, the



```
:- father-in-law (james, anna)
C11: father-in-law (X,Z) :-
     father(X,Y), husband(Y,Z).
C21: father (james, tonny).
C22: father (james, tomson).
C31: husband (tonny, jinny).
C32: husband (tom, anna).
```

(a) An example program (b) The Execution Tree

Figure 2.1: The execution tree of an example

instantiations of all variables trailed after the most recent choice point are erased. In Figure 2.1, Y is trailed at point 2 so that when backtracking occurs at point 4, Y will be restored to be uninstantiated.

2.2 Parallel Execution Models

Parallel logic programming systems differ from one another. The differences range from the types of parallelism exploited to low level implementation techniques. The characteristics of a parallel logic programming system and the methodologies employed in the system are collectively referred to as the *parallel execution model.*

Parallel logic programming systems are crudely featured with the types of parallelism to exploit, inference mechanisms, and the target architectural platform, as explained below.

- Logic programs provide several types of parallelism derived inherently from their language semantics. Most parallel logic programming systems exploit only a subset of those parallelism. Each type of parallelism raises distinct issues in its exploitation. Moreover, the combined exploitation of different types raises additional issues.

- The inference mechanism of logic languages is based on either goal stacking or procedural interpretation. These two mechanisms are in many respects quite different from each other.
- The implementation of logic languages on shared memory multiprocessors is quite different from the one on distributed memory machines. This is mainly because logic programs require the runtime traversal and management of the search tree.

In this section, we use each of the features as a criterion to classify parallel execution models and presents a review of each class.

2.2.1 Classification based on the parallelism

Normally, a Prolog program is represented in an AND/OR tree because it has a simple and regular syntactic structure. The AND/OR tree depicts the parallelism which exists in Prolog programs as shown in Figure 2.2.

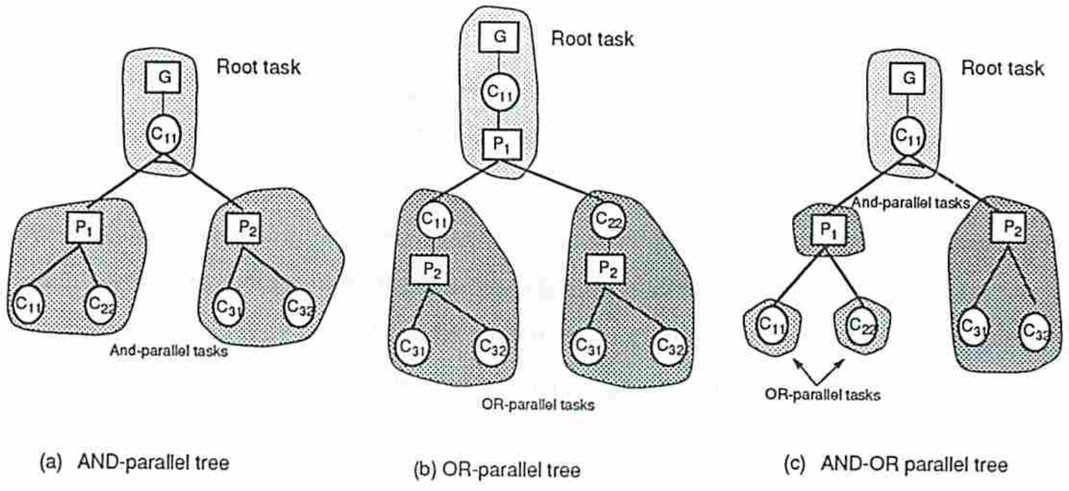


Figure 2.2: Types of Parallelism

The types of parallelism of logic languages which are usually exploited in parallel logic programming systems are OR-, independent AND-, and dependent AND-parallelism. According to the types of the parallelism exploited, we classify parallel execution models as follows.

- OR-parallel model
- Dependent AND-parallel model
- Independent AND-parallel model
- Combined AND/OR-Parallel model

Below, we discuss briefly each model with focus on the characteristics of the parallelism exploited and the main issues pertaining to its implementation.

OR-Parallel Execution Model

In logic programs, the number of clauses which make up a predicate is usually more than one. When the program execution is partitioned for each branch of a predicate node (OR-branch), *i.e.*, for each alternative clause, the parallelism exploited is known as OR-parallelism. The task (OR-task) which executes one of the OR-branches can continue with the next goal in the parent's clause. In Figure 2.2 (b), the task completing the first clause of predicate `father/2` continues with the next goal `husband(Y,Z)`, with Y then instantiated to value `tonny`. The results are passed down the execution tree and the final solutions are available at the leaf tasks, *e.g.,* C_{31} and C_{32} in Figure 2.2 (b).

OR-parallel tasks may share a variable which appears in the head of an ancestor node. When a binding is made to the variable by an OR-task, the bindings must be effective only for the task. The main challenge in the exploitation of OR-parallelism is thus to resolve the binding conflicts in a space and time-efficient manner. For example, the OR-tasks of the goal `father(X,Y)` might attempt to bind Y at the same time. In this case, each of these OR-subtrees must maintain a separate binding environment, as we will see further in chapter 3.

The example OR-parallel models are the Muse [7] and the Aurora [15]. These models are featured with the multiple binding environments. The proposals of the binding environments are the Binding Array [79], Argonne-SRI Model [81], Manchester-Argonne Model [80], Hash Window Method [13], Argonne Model [14], ECRC's PEPSys [82], Virtual Memory Hashing Windows [33], Delphi Model [9], Randomized Method [48], Version Vector Scheme [44], BC-Machine [4], Virtual Memory Binding Arrays model [33], Kabu-Wake Model [58], Directory Tree Method [19], Environment Closing Method [24, 53], Time-Stamping Model [74], and Variable Importation Scheme [60].

AND-Parallel Execution Model

When program partitioning is made at a clause node and calls to the subgoals are executed in parallel, the parallelism exploited is known as AND-parallelism. Figure 2.2 (a) depicts a tree representation of parallel tasks partitioned according to AND-parallelism, where tasks enclosed with a shadowed circle indicate a spawned process. If the subgoals executed in parallel may share any variables, the AND-parallelism is called as *dependent* AND-parallelism; otherwise, it is referred to as *independent* AND-parallelism.

Dependent AND-parallelism: The main difficulty with dependent AND-parallelism is the problem of inconsistent binding. It occurs when more than one AND-subtrees being executed in parallel attempt to bind the same variable differently with their own values, (*e.g.,* variable Y in Figure 2.2). The key issue is thus how to maintain consistent bindings across all *dependent* variables. The most effective approach makes consumer subgoals suspend, provided the consumer subgoals attempt to bind a dependent variable. Later, it awakes the suspended subgoals when the dependent variable is instantiated. The relevant issues include how to determine the producer and the consumer instances for a given dependent variable and how to suspend and to awake the suspended subgoals. The example dependent AND-parallel models are the Andorra Model [42], Pandora [11], and P-Prolog [83].

Independent AND-parallelism: Independent AND-parallelism is easier to implement than dependent AND-parallelism, because the subgoals being executed in parallel do not share variables. A problem with independent AND-parallelism is how to detect the dependencies among subgoals. The detection is made either at compile time [16] or at runtime [28, 59]. The example independent AND-parallel models are Conery's abstract parallel implementation [26], Restricted AND-parallel (RAP) model [28, 65], and AND-Parallel Execution (APEX) model [59].

As the set of subgoals associated with AND-parallelism is in a conjunctive relation, the result of each subgoal will affect the success or failure of the clause. Some relevant issues are as follows:

1. how to keep track of the success/failure of individual literals, and
2. how to determine when a clause fails as a whole.

Combined AND/OR-Parallel Execution Model

When OR-parallelism is combined with AND-parallelism, the results of the OR-tasks may be passed back to the parent AND-task. In Figure 2.2 (c), goals `father(X,Y)` and `husband(Y,Z)` are executed in AND-parallel. OR-tasks are then spawned to execute the clauses of `father(X,Y)` in OR-parallel. The results of these OR-tasks are passed back to the parent task. The OR-tasks do not proceed to the next `husband(Y,Z)` because it is already being executed by an AND-task.

The combined AND/OR parallel models have usually attempted to use either independent AND- and OR-parallelism or independent AND-, dependent AND-, and OR-parallelism. The former approach includes the Pepsys model [82], the AO-WAM [39], the ROPM [52], the ACE [36], and the PBA models [40], while the latter approach includes the IDIOM [36].

2.2.2 Classification based on the inference method

A search tree of a logic program best represents the amount and structure of computation belonging to the logic program. Each node in a search tree stands for a unit of computation and edges stand for the execution order between the units. A number of logic OR-parallel execution models follow the WAM approach, with each processor making a depth-first traversal of some portion of the search tree. We refer to continuous search path traversed by each depth-first search as a *thread.* The parallel execution models based on this approach will be called *thread based* models.

Some parallel execution models are drawn from the procedural interpretation of logic programs. Each clause is viewed as a procedure and a predicate call to a subgoal is manipulated as a procedure invocation. These models are usually implemented by processes which cooperate each other. The search tree is viewed as a process tree, thereby, the parallel execution models based on this approach will be called *process-based* models.

Below, we present the advantages and disadvantages of each approach along with a review of existing implementations.

Thread-based execution models

The main advantage of the thread-based model is to be able to benefit from optimizing techniques developed for sequential execution. Compared with the process-based model, the thread-based model can achieve relatively high absolute single processor performance. Therefore, for reasons of the high single processor performance, most parallel implementations, made particularly on shared memory multiprocessors, adopt this approach. They are the Muse [7], the Aurora [15], the ECRC's PEPSys [82], the Andorra Model [42], the Pandora [11], the P-Prolog [83], the AO-WAM [39], and the ACE [36].

However, these models are not appropriate for message passing architectures. The main reason for this is that OR-parallel execution entails a large amount of fine-grain remote accesses on such architectures. In order to avoid remote accesses, some implementations adopt stack copying. The problem with this approach is that the cost of remote scheduling is very high and the system utilization becomes very low because processors frequently remain idle waiting for the completion of stack copying operations.

Process-based execution models

In process-based execution models, the traversal of the search tree is different from the one in thread-based models. This is because of the procedural interpretation of logic programs; a node created for a subgoal is re-visited after the execution of all the descendents nodes involved in the computation of the subgoal, and then the node for the next subgoal is executed. As a result, a special management of the biding environment is needed both in the forward execution and in the backward execution. In the forward execution, which corresponds to expansion of the process tree, the binding context must be constantly insulated to ensure closed environments at every level in the tree. In backward execution, the back-unification is repeated at every level toward higher levels.

This model is very suitable for distributed implementation because a closing operation made for every unification and back-unification of each OR-task results in the elimination of the remote accesses. One critical drawback of this model is that the single thread performance is very low. This is because within an intra-PE thread which is executed locally on a PE, all the OR-tasks are subject to the closing operation. Moreover, in the distributed implementation, the closed operation puts a limitation on the determination of the grain size. This limitation is drawn from the property that the back-unification of a clause must be done on the same PE where the unification takes place.

The example process-based execution models are the AND/OR process model developed by Conery[23], the ROPM (Reduced Or-Parallel Model) by Kale [52], and the OPAL machine by Conery [51]. In these models, a process created for each goal communicates via messages bindings and control information with other processors in order to finally produce a solution to the top-level query. The main drawback of these models is high overhead caused by the creation and management of processes. Moreover, it is quite difficult to support for stream-AND parallelism in the models.

Data-Driven Approaches

A number of research projects have attempted to benefit from the potential of the data-flow computation model in the parallel processing of logic programs[34]. These approaches can also be classified as process-based models because most of them execute programs under the procedural interpretation.

Bic [12] presented a model for the parallel interpretation of logic programs based on the idea of presenting logic programs as graphs and graph templates in which resolution is viewed as a graph matching. A predicate is represented in a directed arc connecting two nodes that map the arguments of the predicate. Although the implementation exposes a high degree of parallelism on various levels, it is not general enough.

Hasegawa and Amamiya [43] presented an interpretive execution scheme based on a proof tree model. Eager evaluation and lazy evaluation were analyzed to enhance OR-parallel execution as well as to control activation. The approach is based on an interpreter using a high-level functional language. While functional languages is suitable for data-flow architectures, the execution of logic languages through an interpreter written in a high level language would not be a good option due to high overhead.

Ito [46] presented an execution mechanism on a data-flow architecture both for Prolog and concurrent logic languages. It represents in a data-flow graph the generic functions inherent in the languages, *e.g.,* unification and control of non-determinism. The execution mechanism does not support the garbage collection at its parallel unification and backtracking.

Rawling [69] reports the implementation of a concurrent logic language, the Guarded Horn Clause (GHC), on the CSIRAC II data-flow computer. Although a number of issues such as the consistency problem with parallel head unification remains to be addressed, this work contributes to addressing the feasibility of data-flow architectures for a logic programming language.

As an recent alternative approach, the 3DPAM [49, 50] applies the principles of the data-flow model to the design of parallel models for "conventional" parallel architectures, particularly distributed memory multiprocessors. The 3DPAM model envisions the distributed implementation of Prolog by a systematic interpretation of the AND/OR process model [23] from a data-flow point of view.

The Non-deterministic Data-Flow (NDF) parallel execution model [41] uses the data-flow execution paradigm. It is designed for large-scale parallel architectures, particularly distributed memory multiprocessors. In the model, the operational semantics of logic languages are represented by the Non-deterministic Data-Flow graph. The NDF model includes the following distinguished features. (1) It provides a platform for distributed scheduling on top of the data-driven self-organizing execution principle. (2) Fine-grain parallelism internal to OR-tasks is supported by allowing multiple active tasks on a processing element, along with other types of parallelism such as OR-, independent AND-, and stream AND-parallelism. (3) The model also provides a binding method particularly well adapted to distributed memory systems.

2.2.3 Classification based on the architectural platform

Due to the requirement of the maintenance and traversal of a search tree at runtime, the memory organization of parallel machines affects the complexity of parallel implementations of logic languages. In general, the implementations on parallel architectures with a non single address space are relatively complex and inefficient. Indeed, most parallel logic programming systems are designed specific to either of the above two platforms. Parallel execution models developed for shared memory multiprocessors will be referred to as *shared execution models*, and those developed for distributed memory multiprocessors as *distributed execution models.*

Shared Execution Models

Efforts on the parallel processing of logic programs have mostly been devoted to shared memory multiprocessors. With a view to maximally benefit from the conventional sequential techniques, most of these models adopt thread-based execution. The example shared execution models include the Muse [7], the Aurora [15], the ECRC's PEPSys [82], the Andorra Model [42], the Pandora [11], the P-Prolog [83], the AO-WAM [39], and the ACE [36].

Distributed Execution Models

As opposed to the shared execution models, only a few distributed models have been developed. These models explored a methodology which makes variable accesses be restricted locally on each PE. They have been investigated mostly within the context of process-based execution.

In process based approaches, restricted accesses are achieved by developing some binding environments [24, 53, 55, 60]. The example execution models are the 3DPAM [49], the OM [51], the ROMP[52], and the NDF model [41]. In these models, the binding context has to be constantly insulated to ensure closed environments at every level in the tree.

In thread-based approaches, each PE has the same address spaces which implements the conventional four stacks of the WAM. Besides, each PE executes the program in the sequential WAM-like manner. With this approach, it is very hard to efficiently implement the scheduling to another thread that occurs when a processor completes its thread upon either its success or failure. In support of the scheduling, some implementations import the environment through the environment copying to provide the required environment for the new thread, while some build the environment through the recomputation. For example, the Muse [7] is based on the environment copying and the Delphi model [9] is based on the recomputation.

Chapter 3

Analysis of Static Binding Environments

The binding environment is one of important issues in the parallel implementation of logic programs. While it concerns primarily with OR-parallelism, it also affect the performance of other forms of parallelism such as AND-parallelism. In pursuit of an efficient design of a binding environment, we performed an analysis of binding environments. In the analysis, we provide a definition that allows us to classify the binding environments into two groups: *static* and *dynamic* binding environments. In this chapter, we present the analysis of static binding environments. The presentation includes a framework, which helps us analyze the characteristics of binding environments, and the analysis result of binding environments obtained by using the framework. It also includes the application of the analysis result to existing binding environments.

3.1 Introduction

Traditionally, OR-parallelism has been very important source of parallelism in the design of parallel logic programming systems due to its relatively large granularity and simplicity in its management because OR-parallel tasks are independent of each other. The binding environment, which is a memory model to handle multiple binding problems associated with OR-parallelism, has been one of highly important issues in OR-parallel logic programming systems. Indeed, many binding environments have been invented [18, 25, 40, 81, 82] for efficient OR-parallel execution of logic programs and many researches have been conducted to evaluate their performance [38].

It has been noted that the binding environment becomes more important when logic programming systems exploit parallelism more extensively. It is because binding environments affect directly or sometimes indirectly the ways that the other types of parallelism are exploited. Indeed, Gupta shows that the binding environment has also crucial impacts on the efficient exploitation of AND-parallelism [40]. This indicates that in spite of the previous research effort, we still need to explore the efficient design of binding environments when pursuing the parallel implementations of logic programs on large scale parallel machines.

This chapter presents a comprehensive study on the analysis of binding environments. The study is focused on the uncovering of the entire design space of binding environments and the identification of the performance characteristics of each design. We believe that the study will provide an invaluable insight into the design of future binding environments. For the analysis, we develop a framework. The framework consists of definitions for a set of *attributes* which illustrate the organization of a binding environment. It also has definitions for a set of *generic properties* which illustrate the operational characteristics of a binding environment. Given the attributes, all the set of the combinations represent the entire design space of binding environments, in which each one corresponds to one design of a binding environment.

The analysis of static binding environments is carried out through the following two steps: Firstly, for each generic property, we derive all the conditions which guarantee the property. Each condition is represented by some combinations of particular attributes. Secondly, we derive the constant time conditions for each performance criterion by a set of combinations of generic properties. Combining these two steps, we can represent the constant time conditions of each performance criterion by some combinations of attributes. Therefore, it is possible to identify the characteristics of performance criteria for each binding environment in the entire design space. The analysis results show that no design of *static* binding environments exists which guarantees an ideal OR-parallel system. Moreover, we have identified the constant time characteristics of the performance criteria for the existing binding environments by applying the analysis results.

Gupta and Jayaraman's work is one of the most comprehensive studies on the theoretic analysis of OR-parallel logic programming systems [37]. They provide a proof on the non-existent of ideal OR-parallel implementations. Regarding the identification of performance characteristics, the result of our analysis coincides with that of Gupta and Jayaraman's work and thus identifies the fundamental limitation of OR-parallel implementations of logic languages. However, our study has a different goal from Gupta and Jarayaman's. Our study pursues a practical means which brings an insight into the design of future binding environments. Hence, it is focused on the uncovering of the entire design space of binding environments and the identification of the operational as well as the performance characteristics of each design. More specifically, it provides a framework which illustrates the conceptual as well as physical aspects of the organization of binding environments and their operational aspects. We thus believe that our analysis will provide a theoretic foundation of building more efficient binding environments.

The rest of the chapter is organized as follows. Section 3.2 offers a brief description about some terminology and definitions which we will use in later sections. Section 3.3 presents the framework for the analysis. Section 3.4 presents the analysis and its result. Section 3.5 presents the operational properties and performance characteristics of some existing binding environments obtained from the application of our analysis result. Finally, section 3.6 summarizes the chapter.

3.2 Terminology

A *logic program* is composed of a set of *Horn clauses*. Each clause has the form l :-$l_1, \ldots, l_n$, where l_i is a *literal*. A literal consists of a *predicate name* followed by a parenthesized list of terms. A *term* is defined recursively as a constant or a variable or a function symbol followed by a parenthesized list of terms. In a clause, l is the *head* of the clause, and $l_1, \ldots, l_n$ is the *body*. A clause with an empty body is called a *fact*. A clause that is not a fact is called a *rule*. A set of literals is called a *query*. With respect to a goal literal g, the body of a clause

c, instantiated with the substitution that unifies a literal g and the head of c, becomes a query for Q using clause c, which is frequently called the *resolvent.*

3.2.1 Search tree representation

A logic program P and a query Q can be represented by a tree $T = (N, E)$ in which each node is associated with resolvent R. The root of T is associated with the initial query Q. Each node n has a fixed number of child nodes unless its resolvent R is empty. Each child node corresponds to the clause which is in P and successfully unifies with the first goal in R.

For two nodes n_1 and n_2 in tree $T = (N, E)$, if (n_i,n_j) is in E, then n_i is the parent of n_j and n_j is a child of n_i. A path from n_i to n_j is denoted as $path(n_i, n_j)$. For $path(n_i, n_j)$, n_i is an *ancestor* of n_j and n_j is a *descendent* of n_i. Furthermore, if $i \neq j$, then n_i is a *proper ancestor* of n_j and n_j is a *proper descendent* of n_i. In these cases, precedence relations exist between them and they are represented by either $n_i \preceq n_j$ or $n_j \succeq n_i$ when n_i is an ancestor of n_j and by either $n_i \prec n_j$ or $n_j \succ n_i$ when n_i is a proper ancestor of n_j. A node with no descendent is called a $leaf$. A node n and all its descendents are called a *subtree* of T and n is called the *root* of the subtree.

Each child node n_c of node n is associated with a new resolvent R_{n_c}. The new resolvent is an ordered set of literals obtained by replacing the first literal of R_v with the body goals of the corresponding clause and then instantiating every variable within each literal if the variable is substituted during the unification of the first goal literal with the clause head. Formally, let clause c be h:-$b_1, \ldots, b_n$, n be the node with resolvent R_n, n_c a child of n. If R_n is $\{l_1, \ldots, l_n\}$, the new resolvant R_{n_c} becomes $\{(b_1, \ldots, b_n, l_2, \ldots, l_n)\theta\}$, where θ is the *most general unifier* (mgu) resulting from the successful unification between l_1 and the clause head h. Notice that in the above search tree definition, the *branching factor* is fixed for each node and depends on the logic program P. Therefore, for a query Q with respect to a logic program P, an OR-parallel search tree $T = (N, E)$ is defined such that each node n ($n \in N$) is denoted as $n(V,\sigma, R)$ where V is the local environment variables, σ the substitutions, and R the resolvant.

3.2.2 Terminology

This section explains some terminology and definitions which will be used in later sections. In the definitions, T, V, N, and M indicate respectively the set of threads, the set of variables, the set of nodes, and the set of memory in the system.

Worker, thread, standing node: Parallel execution of a logic program is performed by a set of *workers*. Each worker is a sequential engine with its own private memory and shares some memory with other workers. It computes some portions of the search tree, each in the sequential WAM-like manner. The computation that corresponds to the locus of the search tree traversed during each depth-first search is viewed as a unit of computation and called a *thread*. Note that a thread may include more than one tree paths of a search tree. When a worker is computing a node n, node n is called the *standing* node of the worker.

Local environment, global environment: The *local environment* of a node n, denoted as $l(n)$, is the set of variables that appear in the corresponding clause c. The *global environment* of node n, denoted as $g(n)$, is the set of variables obtained by the union of $l(n_i)$ for all i, where $n_i \preceq n$. Local environment $l(n_i)$ is disjoint with $l(n_j)$ if $i \neq j$.

Owner node, binding node, conditional variable, conditional binding, binding environment: If a variable v belongs to the local environment of a node n_o, n_o is called the *owner node* of v. If v is bound in a node n_b, n_b is called the *binding* node. For a node n, any unbound variable v is called a *conditional variable* if the owner node of v is a proper ancestor of n. A binding made to a conditional variable is called a *conditional* binding. The set of conditional bindings that a thread makes during the execution of $path(n_1, n_2)$ is denoted as $cb(n_1, n_2)$. In this chapter, we will sometimes refer to conditional variables and conditional bindings respectively as "variables" and "bindings" without confusion. In OR-parallel implementation of logic languages, some data structure is usually employed to handle conditional bindings. The data structure and the associated manipulation technique will be called the *binding environment* of the system.

Conditional object, conditional context, conditional environment: When a thread t being executed by a worker w makes a conditional binding for a variable v and stores the conditional binding in a memory cell, the memory cell is called a conditional object of v[1]. A conditional object of v is defined for each thread and is associated with some node, usually either the owner node or the binding node. The set of conditional objects used by a worker w_i is defined as the *conditional context* of w_i and is denoted as cc_i. At the time when node n_2 has been computed by thread t, the set of conditional objects *associated* with node n_i for all i $(n_i \preceq n_1)$ is called the *global conditional context* of n_1 with respect to n_2 and t, and is denoted as $gcc(n_1, n_2, t)$. The set of conditional objects associated with node n_i for all i $(n_0 \preceq n_i \preceq n_1)$ is called the *partial conditional context* between n_0 and n_1 with respect to n_2 and thread t and is denoted as $pcc(n_0, n_1, n_2, t)$. The set of values stored in $gcc(n_1, n_2, t)$ is called the *global conditional environment* of node n_1 with respect to n_2 and t and is denoted as $gce(n_1, n_2, t)$. As an example, let us apply the definitions to the Binding Array scheme [79]. The binding array owned by worker w is defined as the conditional context of w. A conditional object is defined as a memory word in binding arrays. Given a thread t being executed by w, $gcc(n_1, n_2, t)$ becomes the part of the binding array allocated by worker w until the execution of n_1 in t. Because a conditional object of a variable is associated with the owner node, $gcc(n_1, n_1, t)$ and $gcc(n_1, n_2, t)$ are always the same.

Static and dynamic binding environment: In a binding environment, if the representation of a conditional binding in a conditional object is simply its binding value without any semantic information such as the binding thread or the binding node, the binding environment is defined as *static*; otherwise, it is defined as *dynamic*. Under this definition, most of the existing binding environments are static binding environments. The examples are the Binding Array [79], the Hash Window [13], the Closed Environment [25, 53], and the Copy-Based OR-parallel schemes [7]. On the other hand, the Time Stamp [74] is a dynamic binding environment because a conditional binding carries some information on the binding node in the form of a time-stamp.

[1]More precise definition is found in the next section.

3.3 Analysis Framework

In this section, we present the framework which we will use to analyze binding environments. To begin with, we introduce a set of performance criteria which have crucial impact on the performance of parallel logic program systems. We then provide comprehensive discussion on the framework.

3.3.1 Performance criteria

Traditionally, variable access, task creation, and task switching have been recognized as the performance criteria for OR-parallel systems [38]. It is because their cost is normally the major factors which determine the performance of an OR-parallel system. The criteria are characterized as follows. Suppose that a worker w is executing a node n in a thread t.

1. *Variable access* refers to the operation that worker w reads (writes) the binding of a variable from (into) the value cell of the variable.

2. *Task creation* refers to the operation that worker w prepares the global environment of the next node in thread t after computing node n.

3. *Task switching* refers to the operation that worker w takes an unexplored node in a different search path after completing thread t. The task switching always results in the creation of a new thread for worker w. Logically, it corresponds to the worker's move from a node n_1 to another node n_2 in which n_1 and n_2 are not in the same path.

In addition to the above three performance criteria, we introduce another performance criterion which we will call *thread switching*:

> A thread switching refers to an operation that worker w suspends thread t which has been executed by it and then chooses for execution another thread being in suspension.

In systems which support multiple threads, a worker is capable of interleaved execution of multiple threads. OR-parallel logic programming systems have a great opportunity to benefit from multiple thread, particularly in the following cases.

- When a worker encounters some extra-logical or side-effect causing predicate, the worker cannot continue the execution before the branch, which corresponds to the predicate in the search trees, becomes the leftmost. In this case, the worker should either wait until the branch becomes the leftmost or suspend the running thread and then execute another available thread.
- The interleaved execution of multiple threads allows the system to hide communication latency caused by such activities as scheduling in distributed memory multiprocessors and stack copying made between processors with non-uniform memory accesses.

Some recent logic programming systems empirically show that suspension occurs very frequently and efficient support for multiple threads is essential for higher system performance [8]. This indicates that the cost of thread switching is one of important factors which affect the system performance.

Both task switching and thread switching include a worker's move from a node to another. However, it should be noted that task switching always creates a new thread whereas thread switching executes an existing thread without creating a new one.

3.3.2 Attributes of conditional objects

In the implementation of OR-parallel systems, there are a number of issues pertaining to the management of multiple bindings. How to address each of those issues determine various aspects of its operational characteristics and performance. Those aspects include the data representation of a conditional binding, the data structure to store conditional bindings, and the memory organization to implement the data structure.

In the previous section, we introduced a term *conditional object* as the unit of storage which stores a conditional binding in a binding environment. In this subsection, we present the precise definition of a conditional object and then discuss the set of the attributes that illustrate conditional objects with respect to binding environments.

Definition of conditional objects

A conditional object is defined only in association with accesses of conditional variables. The precise definition of a conditional object is as follow:

> Given a conditional variable, the conditional object is the *minimum set of memory cells* each of which *must be always accessed* for *the access of the variable.*

The definition dictates that a conditional object is essentially represented distinctively depending on the memory organization of a binding environment. For example, a conditional object is represented as a single memory word in the Binding Array, as a hash entry in the Hash Window, and as a linked list in the Time Stamp.

The design space of a conditional variable is very wide. For a conditional variable, some systems provide one conditional object which is shared by multiple threads. On the other hand, some systems provide more than one conditional objects such that each thread accesses its own conditional object. When a thread accesses in a node a conditional variable, it must always refer uniquely to a single conditional object. In this regard, the conditional object of a variable v is defined with respect to a node n and a thread t, which is denoted as $co(v, n, t)$ where co is a mapping such that $co : V\text{x}N\text{x}T \rightarrow M$.

It should be noted that a conditional object is defined with respect to variable accesses. When more than one instances of the conditional binding are kept in a system, only the instance that is related with the variable accesses is concerned with the definition of the conditional objects. For example, the Binding Array method keeps two instances for each conditional binding: one in the forward list and the other in the binding array. The values in the forward list are used in scheduling such as backtracking and task switching, while the values in the binding array are used in the variable accesses made during computation such as unification and argument generation. In this case, only the values in the binding array are subject to the definition of conditional objects.

The attributes of conditional objects

The binding environments show different characteristics for the performance criteria. Careful observation on a number of existing binding environments leads us to an insight that given a binding environment, the characteristics are determined by the design of conditional objects in the binding environment. We thus investigated the design space of conditional objects in binding environments and identified that the design is characterized by the following questions:

- Among the worker, the thread, and the node, to which one does a conditional object allocated?
- Which node has the information that a worker will use to locate the conditional object?
- How many instances of a conditional object are allocated along a single search path?
- How many bindings is a conditional object able to contain?
- Is a binding represented by a simple data value or by a combination of a data value and other semantic information?

In order to identify all the possible answers, we defined five attributes with regard to conditional objects. The above questions are then reduced to the problem of value assignment to the attributes. Each case of value assignment for the set of attributes corresponds to one design of a binding environment. The set of all possible cases represents the entire design space of the binding environment. In other words, it corresponds to the design space of the OR-parallel system within the domain of the multiple binding problem. We illustrate each attribute and values assigned to it.

1. Owner

When a conditional variable is bound by a worker during the computation of a node in a thread, the binding is kept in the conditional object allocated in the memory. In the design, the allocation can be made in one of the following three meaningful ways: allocation with respect to the worker, the owner, or the thread. Attribute "owner" illustrates to which one a conditional object is allocated among a worker, a thread, and a node. The three values, **worker**, **thread**, and **node**, are thus defined for attribute "owner". When a conditional object is allocated with respect to the worker (*resp.* thread, or node), it is called that the conditional object is owned by the worker (*resp.* thread, or node).

2. Association

For a given conditional variable, attribute "association" illustrates which node has the information that a worker will use to locate the conditional variable. It has `owner` and `binding` as its values. In the WAM, the name of a variable is the address of a slot in the environment frame to which the variable belongs [78]. Some design of OR-parallel systems maintains a piece of information in the slot which a worker will use to locate the conditional object of the variable. The information is usually either a direct or an indirect pointer to the conditional objects. In this case, it is called that the conditional object is *associated* with the *owner* node because the slot belong to the owner node. If the conditional object is associated with a node other than the owner node, it is called that the conditional variable is *associated* with the *binding* node. This is because the binding node can best represent those nodes that are not the owner node. However, it should be noted that the conditional object of this case is not necessarily associated with the binding node and can be associated with any node except the owner node.

3. Instantiation

For a given conditional variable, some design of a binding environment may allocate more than one conditional objects differently along a single path. In this case, each one is called an *instance* of the conditional object. Provided that an instance is allocated differently for every node of a path, the number of instances may becomes infinite. On the other hand, in some design of a binding environment, the maximum number of instances is always limited to a finite number. Attribute "instantiation" illustrates whether the maximum number of instances is restricted to a finite number or not. Two values defined for attribute "instantiation" are thus **finite** and **infinite**.

4. Capacity

In some design of an OR-parallel system, a conditional object may consist of more than one value cells, where a value cell is a unit of storage which stores a single conditional binding. Attribute "capacity" describes the amount of value cells allowed for a conditional object. Two values, **finite** and **infinite**, are defined for the attribute. In OR-parallel systems in which attribute "capacity" is assigned to value **finite** (*resp.* **infinite**), a finite (*resp.* infinite) number of value cells is allowed for a conditional variable. It should be noted that each value cell in a conditional object does not necessarily keep a binding. For example, in some binding environments [25, 60], even though a conditional object consists of multiple value cells such that one value cell is allocated for each node along the path between the owner and the binding node, only the last value cell may keep the binding, while the others remain empty.

5. Content

In OR-parallel systems, a conditional binding is represented either by just a data value for the binding or by the combination of the data value and some semantic information such as the binding node and the binding worker. If the conditional binding stored in a conditional object is just the data value, the conditional object is called to keep a *static* data. Otherwise, the conditional object is called to keep a *dynamic* data. Attribute "type" illustrates whether a conditional object is designed for *static* or *dynamic* data. Two values, **static** and **dynamic**, are thus defined for attribute "type".

Remarks:

In the rest of the chapter, we use a notation attr-name[value] to represent the assignment of attribute "attr-name" with its value "value". The list of attributes and their values will be denoted as {name1[value1], name2[value2], ...}. When attribute "attr-name" of a conditional object is always set to "value", a conditional object is called to *have* attr-name[value]. For brevity, we use the following shorthand notation: attributes **owner**, **association**, **instantiation**, **capacity**, and **content** are represented respectively by **own**, **ass**, **ins**, **cap**, and **con**, and values **worker**, **thread**, **node**, **owner**, **binding**, **infinite**, and **finite** are represented respectively by **W**, **T**, **N**, **O**, **B**, **I**, and **F**.

3.3.3 Design space of the binding environment

From the above five attributes and its values, 48 different designs of the binding environments are enumerated. A half of them are static binding environments, while the others are dynamic binding environments. This subsection discusses the designs with focus on static binding environments.

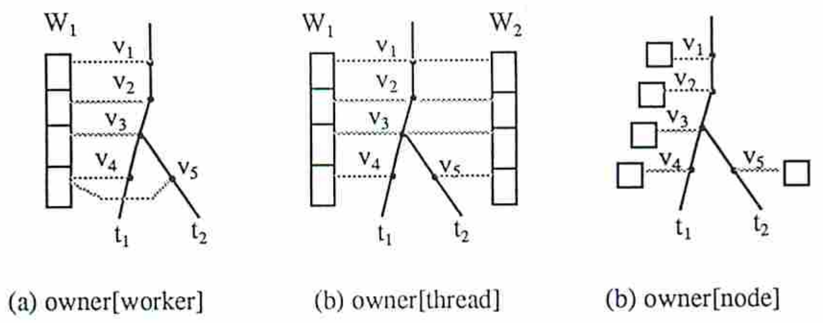


Figure 3.1: Designs resulting from different values of attribute "owner"

Figure 3.1 shows three cases of binding environments which result from different values of attribute "owner". Suppose that the least common ancestor node between t_1 and t_2 is $n_{c_{12}}$.

- In a binding environment with **owner[worker]** (Figure 3.1 (a)), a conditional context is provided for each worker. If worker w_1 executes both t_1 and t_2, the conditional context is used for t_1 when w_1 executes t_1, whereas it is used for t_2 when w_2 executes t_2. Notice that global conditional contexts $gcc(n_{c_{12}}, n_{c_{12}}, t_1)$ and $gcc(n_{c_{12}}, n_{c_{12}}, t_2)$ are the same.
- In a binding environment with **owner[thread]** (Figure 3.1 (b)), a conditional context is provided for each thread, *i.e.*, $gcc(n_{c_{12}}, n_{c_{12}}, t_1) \cap gcc(n_{c_{12}},$

$n_{c_{12}}, t_2) = \phi$. Notice that when worker w executes both t_1 and t_2, interleaved execution of t_1 and t_2 can be achieved more efficiently in this case.

- In a binding environment with **owner[node]** (Figure 3.1 (c)), a global conditional context of a node is the collection of conditional objects associated with all the ancestor nodes. Given two threads, the global conditional context for the shared part of the path is the same, *i.e.*, $gcc(n_{c_{12}}, n_{c_{12}}, t_1) = gcc(n_{c_{12}}, n_{c_{12}}, t_2)$ as the case of **owner[worker]**. However, conditional contexts are defined differently for the private parts, *i.e.*, $gcc(n_{c_i}, n_{c_i}, t_1) \cap gcc(n_{c_j}, n_{c_j}, t_2) = \phi$ ($n_{c_{12}} \prec n_i$ and $n_{c_{12}} \prec n_j$).

Attribute "instantiation" and "capacity" have values either **finite** or **infinite**. As an example, in a binding environment with **{inst[F],cap[F]}**, a finite number of conditional objects, each of which consists of a finite number of value cells, are allocated in a single path. Because the size of a search tree is not limited to a finite number in terms of its node depth and the number of branches, any finite number other than one is not normally a reasonable choice. Therefore, the discussion in this chapter will be restricted to the case of "one" when the attribute value is **finite**.

Figure 3.2 (a),(b),(c), and (d) depict the four cases of value assignment. In the figure, a box denoted by a solid line stands for a conditional object and a square separated by a dotted line inside a conditional object represents a value cell. A dotted line between a conditional object and a node indicates the *association*. According to the value of "association", it can be either the owner node or the other nodes such as the binding node. With association[owner], the above four cases are briefly explained.

- In binding environments with **{ins[F],cap[F]}** (Figure 3.2 (a)), a conditional variable has only one value cells and only a single instance is available for a path.
- In binding environments with **{ins[F],cap[I]}** (Figure 3.2 (b)), a conditional object is capable of keeping all the bindings made from descendent nodes because its size is not limited to a finite number. For correct execution, each binding must contain additional information regarding the binding node or

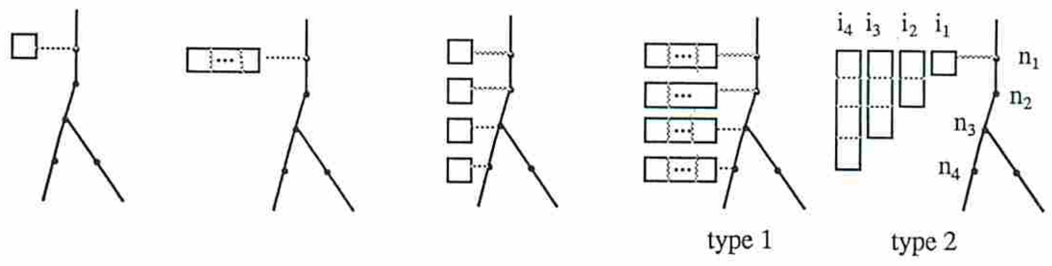
the binding thread. Therefore, the resulting binding environment is usually dynamic.

- For a given conditional variable, each node has an instance of the conditional object in binding environments with **{ins[I],cap[F]}** (Figure 3.2 (c)). The association of each instance does not concerned with either the owner or the binding node of the conditional variable. As a matter of fact, this is viewed as a special case that the association of each instance becomes the node to which the instance belongs. Notice that it operationally corresponds to the importation of the variable from a parent node to a child node [60], because the child node becomes the owner node for the imported variable.
- Two different types are possible with with **{ins[I],cap[I]}** (Figure 3.2 (d)). Due to **{ins[I]}**, given a conditional variable, an instance of a conditional object is defined for each node along a path. Type 1 is the same as the case (c) except that a conditional object can have an infinite number of cells. Type 2 is different from type 1 in that all the instances are associated with the original owner node even though each of them is assigned to a different node. One meaningful design of type 2 is that the conditional object of a child node has one more cell than that of the parent node. For example, Figure 3.2 (d) shows four instances defined respectively for nodes n_1, n_2, n_3, and n_4. In this type, only a single instance will have a binding, because the multiple instances are for a single path. Physical implementation of these instances can be made by using a single array such that each cell is assigned to a node and the instance assigned to a node amounts to the portion of arrays which start from the first cell and ends at the cell allocated for the node. The Variable Importation scheme [60] is one salient example of this type. In type 1, each conditional variable is imported into each child node. Under the principle of importation, the only reasonable way to manage the binding is to make the binding be associated with the binding node. This makes the conditional object always keep at most one binding, even though it has an infinite number of cells. As a result, type 1 is always reduced to the case of Figure 3.2 (c) in terms of its functionality and operational principle.

No	Attributes				No	Attribute				No	Attribute			
	Ow	As	In	Ca		Ow	As	In	Ca		Ow	As	In	Ca
1	W	O	F	F	9	T	O	F	F	17	N	O	F	F
2				I	10				I	18				I
3			I	F	11			I	F	19			I	F
4				I	12				I	20				I
5		B	F	F	13		B	F	F	21		B	F	F
6				I	14				I	22				I
7			I	F	15			I	F	23			I	F
8				I	16				I	24				I

Table 3.1: The classes of OR-parallel systems with static binding environments: "No" indicates the class number and "Ow", "As", "In", and "Ca" stand for attributes Owner, Association, Instantiation, and Capacity, respectively.

In the rest of the chapter, the case of **{ins[I],cap[I]}** is always regarded as type 2.



(a) {ins[F],cap[F]} (b) {ins[F],cap[I]} (c) {ins[I],cap[F]} (d) {ins[I],cap[I]}

Figure 3.2: A pictorial representation of four designs of binding environments with respect to attributes **instantiation** and **capacity** when attribute **finite** is 1.

3.3.4 The properties of conditional contexts

This section presents a set of properties which we identified with regard to conditional contexts. These properties represent the operational aspects of the generic operations (variable accesses, task creation, task switching and thread switching) as well as the aspects of the physical organization of conditional contexts.

Intra-thread properties

Depending on the organization of conditional contexts, the bindings of conditional variables may sometimes modify the environment of the ancestor nodes. If a binding modifies the environment of the ancestor nodes, the following two operations are entailed: (i) the deinstallation of the conditional bindings made along the path between the least common ancestor node and the standing node, and (ii) the installation of the conditional bindings made along the path between the least common ancestor node and the destination node. We therefore identified a property which illustrates the effect that a conditional binding makes on the contents of a conditional context.

Definition 3.3.1 (Protectedness) *Given a conditional context, when a node n_1 and n_2 ($n_1 \prec n_2$) are executed in a thread t, if conditional bindings made in node n_2 are not written in $gcc(n_1, n_1, t)$, the conditional context is defined to provide protectedness.*

The access of a conditional binding is logically viewed to have the following two steps: locating the conditional object and accessing the value cell of the conditional binding in the conditional object. Associated with these steps, the following two properties are defined with respect to conditional contexts.

Definition 3.3.2 (Deterministic-locatedness) *Given a conditional context, if the conditional object of a variable is always located without involving with any search of infinite sets, the conditional context is defined to provide deterministic-locatedness.*

Definition 3.3.3 (Deterministic-access) *Given a conditional context, if the conditional binding of a variable is accessed from its conditional object without* *involving with any search of infinite sets, the conditional context is defined to provide deterministic-access.*

The cost of task creation depends on how to efficiently create the necessary environment for the new node. In some conditional context, the creation of the environment may sometimes involve with an infinite set of conditional objects. Associated with task creation, another property is defined as follows.

Definition 3.3.4 (Inheritedness) *Given a conditional context, when node n_1 and its child node n_2 are executed in a thread t, if the difference between $gcc(n_1, n_1, t)$ and $gcc(n_2, n_2, t)$ is always a finite set, the conditional context is defined to provide inheritedness.*

A conditional context is physically organized by a set of conditional objects. The way that a worker manages the conditional bindings is influenced by the way that the conditional objects are organized in the conditional context. Associated with the organization, a property is defined with respect to conditional contexts.

Inter-thread properties

Definition 3.3.5 (Physical-sharedness) *Given two threads t_1 and t_2 working on a conditional context, regardless of whether t_1 and t_2 are executed concurrently or one thread is executed after the completion of the other, if $gcc(n, n, t_1)$ is equal to $gcc(n, n, t_2)$ for any common ancestor node n between t_1 and t_2, the conditional context is defined to provide physical-sharedness.*

Most recent parallel logic systems preserve the sequential semantics of logic languages completely. They thus support side effects and extra-logical predicates as well as the pure logic predicates. In such systems, when a worker encounters a side effect or an extra-logic predicate, the worker should not execute them until the branch becomes the leftmost. In the mean time, (i) the worker either busy-waits or (ii) switches to another thread to execute, suspending the previous thread. In either case, the system should pay some overhead such that in case (i), the system utilization will lower due to worker's idling and in case (ii), the system may discard the environment of the previous thread because of the switching. With respect to conditional contexts, another property is defined which will characterize the efficiency of the thread switching.

Definition 3.3.6 (Preservedness) *Given a conditional context, if a worker performs thread switching from a thread t_1 to another thread t_2, without causing any change of $gce(n_1, n_1, t_1)$, and also it does not destroy $gce(n_1, n_1, t_1)$ while executing thread t_2, the conditional context is defined to provide preservedness.*

3.4 Analysis Results

This section presents the analysis result. In the analysis, we derive the conditions which will satisfy each property of conditional contexts. We also derive the conditions which will satisfy the constant time requirement for each performance criterion. Conditions for the properties of conditional contexts are represented by a set of attribute-value pairs, whereas conditions for the constant time requirement of each performance criterion are represented by a combination of the properties defined with respect to conditional contexts.

3.4.1 Generic properties

Proposition 1 (Protectiveness) *For the following combinations of attribute and value pairs,*

Case 1: {association[`binding`]}
Case 2: {association[`owner`],instance[`infinite`]}
Case 3: {association[`owner`],instance[`finite`]}

it is always possible to build a conditional context, which provides protectedness, for case 1 and 2, whereas it is not possible for case 3.

(Proof) Let n_o and n_b be respectively the owner and the binding node of variable v. Suppose that they are executed by thread t.

- Case 1: Due to $n_o \prec n_b$ and association[`binding`], $co(v, n_b, t) \notin gcc(n_o, n_o, t)$ and co$(v, n_b, t) \in gcc(n_b, n_b, t)$. Therefore, the binding of v to be stored in $co(v, n_b, t)$ does not change $gcc(n_o, n_o, t)$.

- Case 2: Due to $n_o \prec n_b$ and association[owner], $co(v, n_o, t) \in gcc(n_o, n_o, t)$. With instance[infinite], more than one conditional objects are defined in path(n_o,n_b). (i) Suppose that the attribute "capacity' is finite. As for variable v, a new variable is created in each node of $path(n_o, n_b)$ and a condition object is also created for the new variable, as explained in Figure 3.2 (c). In node n_b, the binding of v is stored in the new conditional object $co(v, n_b, t)$ ($co(v, n_b, t) \notin gcc(n_o, n_o, t)$). (ii) Suppose that the attribute "capacity' is infinite. As explained in Figure 3.2 (d), a conditional object is created in each node of $path(n_o, n_b)$. For a parent node n_1 and a child n_2 in $path(n_o, n_b)$, $co(v, n_2, t)$ is obtained by appending a cell to $co(v, n_1, t)$ and is associated with the original owner node. If a binding is made in n_2, it is stored in the cell. Therefore, the binding of v made in the binding node n_b is always stored in the new conditional object $co(v, n_b, t)$ ($co(v, n_b, t) \notin gcc(n_o, n_o, t)$). From in (i) and (ii), the binding of v in node n_b is not stored in $gcc(n_o, n_o, t)$.
- Case 3: Let a finite number x be the number of instantiations. Due to association[owner] and instance[finite], in $path(n_o, n_b)$ at most x instantiations can be defined for a conditional object and they are respectively associated with x nodes in $path(n_o, n_b)$. When the difference between the depth of the binding node and that of the owner node is larger than x, the binding made in the binding node must be stored in an instantiation which belongs to $gcc(n, n, t)$, where n is an ancestor node of $b(v)$. Therefore, it is not possible to make a conditional context which provides protectedness.

Proposition 2 (Inheritability) *For the following combinations of attribute and value pairs,*

Case 1: {instance[finite]}
Case 2: {instance[infinite]}

it is always possible to build a conditional context, which provides inheritedness, for case 1, whereas it is not possible for case 2.

(Proof) Let n_2 be the parent node of n_1 and S be the set of conditional variables for n_2.

- Case 1: The difference between $gcc(n_1, n_1, t)$ and $gcc(n_2, n_2, t)$ is the union of the local environment of n_2 and the conditional objects allocated both in n_1 and n_2. Because both the local environment and the number of conditional objects allocated in each node are finite sets, the difference is always a finite set.
- Case 2:
  Let S be the set of unbound conditional variables when node n_1 is computed. Because of instance[`infinite`], for each variable v in S a new conditional object is created as for $co(v, n_2, t)$ in node n_2. Because S is an infinite set and $co(v, n_2, t)$ is always different from $co(v, n_1, t)$ for every variable v ($v \in S$), $gcc(n_1, n_1, t) \cup gcc(n_2, n_2, t)$ is not a finite set. □

Proposition 3 (Deterministic-locatedness) *For the following combinations of attribute and value pairs,*

Case 1: {association[`owner`]},
Case 2: {association[`binding`]},

it is always possible to build a conditional context, which provides deterministic-locatedness, for case 1, whereas it is not possible for case 2.

(Proof) Assume that a thread t accesses in a node n a conditional variable v.

- Case 1: Regardless of whether v is a bound or an unbound variable, t can locate $co(v, n_o, t)$ without engaging any search of an infinite set since $co(v, n_o, t)$ is associated with the owner node.
- Case 2: (i) If the variable v is not yet bound, t has to scan $pcc(n_o, n, n, t)$ before it finds out that v is not yet bound. (ii) If the variable v is bound in n_b, t has to scan $pcc(n_o, n_b, n, t)$ before it can locate $co(v, n_b, t)$. Because $pcc(n_o, n, n, t)$ and $pcc(n_o, n_b, n, t)$ are infinite sets in logic languages, t must always search a finite set in order to locate $co(v, n_b, t)$. □

Proposition 4 (Deterministic-access) *For the following combinations of attribute and value pairs,*

Case 1: {dimension[`finite`]}
Case 2: {dimension[`infinite`],association[`binding`]}
Case 3: {dimension[`infinite`],owner[`worker|thread`],instance[finite]}
Case 4: {dimension[`infinite`],association[`owner`],instance[infinite]}
Case 5: {dimension[`infinite`],owner[`node`],association[`owner`],instance[finite]}

it is always possible to build a conditional context, which provides deterministic-access, for case 1, 2, and 3, whereas it is not possible for case 4 and 5.

(Proof)

- Case 1: Due to dimension[`finite`], a conditional object consists of a finite number of cells; the access of a cell in a conditional object never engages a search of an infinite set.
- Case 2: A conditional object may have an infinite number of cell due to dimension[`infinite`]. Because the conditional object is associated with $b(v)$ due to association[`binding`], it is accessed by some descendent nodes of $b(v)$. The variable will not be bound in the descendent nodes of $b(v)$ on account of the single assignment property of logic languages. Hence, it is possible to always put the binding in the first cell, that conforms to association[`binding`]. In this case, the access of the conditional binding does not include any search of an infinite set.
- Case 3: In spite of capacity[`infinite`], at most one binding is stored in a conditional object, provided attribute "owner" is either `worker` or `thread` and "instantiation" is `finite`. In this case, it is always possible to store the binding in the first cell. Hence, the access of the conditional binding does not involve with any search of an infinite set.
- Case 4: A conditional object may have an infinite number of cells due to dimension[`infinite`]. As explained in Figure 3.2 (d), a conditional variable is stored in a cell which cannot be determined beforehand. Therefore, the access of the conditional binding in a conditional object involves with a search of an infinite set.

- Case 5: {dimension[`infinite`],owner[`node`],association[`owner`], instance[finite]} As discussed in section 3.4, this case always results in a dynamic binding environment. Although a dynamic binding environment is beyond the scope of this chapter, a brief discussion is offered to illustrate the Time Stamp method. Suppose that a binding of a variable is placed in the next free cell of the conditional object with some semantic information which will be used to identify the binding node. Because a conditional object for a variable v has {association[`owner`],instance[`finite`]}, it is accessed by descendent nodes of $o(v)$. Moreover, the variable can be bound in the descendent nodes of $o(v)$ other than those of b(v). Because the number of the descendent nodes of o(v) is not finite in logic languages, it is not possible to use a specific cell as does in case 2. The access of a conditional binding always involves with a search of an infinite set. □

Proposition 5 (Physical-sharedness) *For the following combinations of attribute and value pairs,*

Case 1: {owner[`node`]}
Case 2: {owner[`worker`]}
Case 3: {owner[`thread`]}

it is always possible to build a conditional context, which provides physical-sharedness, for case 1, whereas it is not possible for case 2 and 3.

(proof) Given a node n and its two child nodes n_1 and n_2, suppose that threads t_1 and t_2 execute respectively n_1 and n_2. Consider a conditional variable v ($o(v) \preceq n$) and its conditional object.

- Case 1: Because $co(v, n, t_1)$ and $co(v, n, t_2)$ are defined as the same conditional object, $gcc(n, n, t_1)$ is always equal to $gcc(n, n, t_2)$.
- Case 2 and 3: Suppose that workers w_1 and w_2 execute respectively t_1 and t_2. Because $co(v, n, t_1)$ and $co(v, n, t_2)$ are defined as different conditional objects, $gcc(n, n, t_1)$ is not equal to $gcc(n, n, t_2)$. □

Proposition 6 (Preservedness) *For the following combinations of attribute and value pairs,*

Case 1: {owner[`node`]}
Case 2: {owner[`thread`]}
Case 3: {owner[`worker`]}

it is always possible to build a conditional context, which provides preservedness, for case 1, whereas it is not possible for case 2 and 3.

(Proof) Consider a thread switching from a node n_1 of a thread t_1 to a node n_2 of a thread t_2. Let the least common ancestor node be n_c.

- Case 1: Because of {owner[`node`]}, $gcc(n_1, n_1, t_1) \cap gcc(n_2, n_2, t_2) = gcc(n_c, n_c, t_1)$. For correct execution, the system must be organized such that $gcc(n_c, n_c, t_1)$ is not written by either t_1 or t_2 in any node n ($n_1 \preceq n$ and $n_2 \preceq n$). Therefore, $gcc(n_1, n_1, t_1)$ is preserved while thread t_2 is executed.

- Case 2: Because of {owner[`thread`]}, $gcc(n_1, n_1, t_1) \cap gcc(n_1, n_2, t_2) = \phi$; $gce(n_1, n_1, t_1)$ is preserved while thread t_2 is executed.
- Case 3: Suppose that a worker is executing t_1 and t_2 in a interleaved fashion. For a conditional variable v $(o(v) \preceq n_c)$, $co(v, o(v), t_1)$ and $co(v, o(v), t_2)$ are always defined the same. Some conditional objects in $gcc(n_1, n_1, t_1)$ are thus written by t_2; $gce(n_1, n_1, t_1)$ is not preserved while thread t_2 is executed. □

3.4.2 Performance criteria

In this subsection, we derive the constant time condition for each performance criterion by using the properties defined for conditional contexts. In the discussion, a binding environment B implementing some conditional context cc is denoted as $B(cc)$.

Lemma 1 *The variable accesses are constant time operations in an OR-parallel logic programming system with a binding environment $B(cc)$ iff cc provides both deterministic-locatedness and deterministic-access.*

(Proof) Variable accesses occur either for private or for conditional variables. (i) The bindings of private variables can always be accessed in a constant time because the value cells can be directly located by its name. (ii) If cc provides deterministic-locatedness and deterministic-access, the variable accesses of conditional variables do not involve with any search of an infinite set; otherwise, they involve with a search of some infinite set. From (i) and (ii), variable accesses are always constant time operations, iff cc provides deterministic-locatedness and deterministic-access. □

Lemma 2 *The task creation is always a constant time operation in an OR-parallel logic programming system with a binding environment $B(cc)$, iff cc provides inheritedness.*

(Proof) Suppose that node n_1 is the parent of node n_2 and thread t is about to create a task for n_2 after completing n_1. Normally, task creation for n_2 consists

	Conditional Attributes				Operational Properties						
	All	Ass	Ins	Dim	PRT	IN	DL	DA	PS	PRS	Example
1			F	F	n^3	y^1	y^1	y^1	n^2	n^2	VA, VV, CB
2		O		I	n^3	y^1	y^1	y^3	n^2	n^2	
3			I	F	y^2	n^2	y^1	y^1	n^3	n^2	
4				I	y^2	n^3	y^1	n^4	n^2	n^2	
5	W		F	F	y^1	y^1	n^2	$y^{1,2}$	n^2	n^2	
6		B		I	y^1	y^1	n^2	y^2	n^2	n^2	
7			I	F	y^1	n^2	n^2	$y^{1,2}$	n^2	n^2	
8				I	y^1	n^2	n^2	y^2	n^2	n^2	
9			F	F	n^3	y^1	y^1	y^1	n^2	y^1	
10		O		I	n^3	y^1	y^1	y^3	n^2	y^1	
11			I	F	y^2	n^2	y^1	y^1	n^2	y^1	
12				I	y^2	n^3	y^1	n^4	n^2	y^1	
13	T		F	F	y^1	y^1	n^2	$y^{1,2}$	n^2	y^1	
14		B		I	y^1	y^1	n^2	y^2	n^2	y^1	
15			I	F	y^1	n^2	n^2	$y^{1,2}$	n^2	y^1	
16				I	y^1	n^2	n^2	y^2	n^2	y^1	
17			F	F	x	x	x	x	x	x	
18				I	n^3	y^1	y^1	n^5	y^1	y^1	TS
19	N	O	I	F	y^2	n^2	y^1	$y^{1,2}$	y^1	y^1	CE, RCE
20				I	y^2	n^2	y^1	n^4	y^1	y^1	VI
21			F	F	y^1	y^1	n^2	$y^{1,2}$	y^1	y^1	HW
22				I	y^1	y^1	n^2	y^2	y^1	y^1	
23		B	I	F	y^1	n^2	n^2	$y^{1,2}$	y^1	y^1	
24				I	y^1	n^2	n^2	y^2	y^1	y^1	

Table 3.2: Attributes and the properties: PRT, IN, DL, DA, PS, and PRS stand for respectively protectedness, inheritedness, deterministic-locatedness, deterministic-access, physical-sharedness, and preservedness.

of the creation of the global conditional environment $gce(n_2, n_2, t)$ and the local environment $l(n_2)$. (i) Because $gcc(n_2, n_2, t)$ is an infinite set in logic languages, the only way to create $gce(n_2, n_2, t)$ in a constant time is to reuse $gce(n_1, n_1, t)$ and to create only the difference between $gce(n_1, n_1), t)$ and $gce(n_2, n_2, t)$ in a constant time. If cc provides inheritedness, the difference between $gcc(n_1, n_1, t)$ and $gcc(n_2, n_2, t)$ is always an finite set; otherwise, it is not an infinite set. (ii) Creation of a local environment is alway a constant time operation since the local environment $l(n_2)$ is a finite set in logic languages. From (i) and (ii), the task creation is always a constant time operation, iff cc provides inheritedness. □

Lemma 3 *A task switching is always a constant time operation in an OR-parallel logic programming system with a binding environment $B(cc)$, iff cc provides protectedness and physical-sharedness.*

(Proof) Suppose that a worker performs a task switching from a node n_1 in a thread t_1 to a child of a node n_2 in a thread t_2 and in consequence of the task switching, thread t_n is created for the worker. Let the least common ancestor of the two nodes be node n_c. The task switching is reduced to the problem of creating $gce(n_1, n_2, t_n)$ on $gcc(n_2, n_2, t_n)$ to be the same with $gce(n_1, n_2, t_2)$. Note that it is not possible to create $gce(n_2, n_2, t_n)$ in a constant time by copying $gce(n_2, n_2, t_2)$ because $gcc(n_2, n_2, t_2)$ is an infinite set.

We will divide the creation of $gce(n_2, n_2, t_n)$ into two parts: $gce(n_c, n_c, t_n)$ and the remaining part. (i) Suppose that cc provides physical-sharedness. Thread t_1 shares with t_2 the global conditional context of n_c and also t_n shares with t_n the global conditional context of n_2, *i.e.*, $gcc(n_2, n_2, t_n) = gcc(n_2, n_2, t_2)$. Therefore, the creation of $gce(n_c, n_c, t_n)$ is inherently a constant time operation. However, cc must always provide protectedness; otherwise, it is impossible to construct a binding environment which functions correctly. (ii) Suppose that cc does not provides physical-sharedness. Threads t_1 and t_2 do not share a conditional context, and thus t_n will have its own conditional context. There are two ways to create $gce(n_2, n_2, t_n)$, either (a) by newly allocating $gcc(n_2, n_2, t_n)$ and copying the values stored in $gcc(n_2, n_2, t_2)$ into it, or (b) by converting $gce(n_c, n_2, t_1)$. In case (a), the creation takes a non-constant time, because $gcc(n_2, n_2, t_2)$ is an infinite set. In case (b), the conversion of $gce(n_c, n_2, t_1)$ into $gce(n_2, n_2, t_n)$ consists of the following parts:

1. (deinstallation:) repealing $cb(n_{c_1}, n_1)$ from $gcc(n_c, n_c, t_1)$, and
2. (installation:) installing $cb(n_{c_2}, n_2)$ into $gcc(n_2, n_2, t_1)$.

Now that $cb(n_{c_1}, n_1)$ and $cb(n_{c_2}, n_2)$ in the above steps are infinite sets and the conversion thus always takes a non-constant time, the task switching always takes a non-constant time. From (i) and (ii), the sufficient and necessary condition for a constant time task switching is that *cc* must provide physical-sharedness and protectedness. □

Lemma 4 *A thread switching is always a constant time operation, if cc provides preservedness.*

(Proof) Assume that a worker w suspends the execution of a thread t_1 after computing a node n_1.

(i) Consider a thread t_2 being suspended and to be computed from a node n_2 by any worker. Suppose that w switches from t_1 to t_2. If *cc* provides preservedness, $gce(n_1, n_1, t_1)$ is preserved without involving with any non-constant time operation both at thread switching and during the execution of thread t_2. If b does not provide preservedness, the contents of $gcc(n_1, n_1, t_1)$ must be saved in some other place. Now that $gcc(n_1, n_1, t_1)$ is an infinite set, the thread switching becomes a non-constant time operation. Therefore, thread switching is always a constant time operation, iff *cc* provides preservedness. (ii) Suppose that there is no thread being suspended and thus a worker must take an unexplored child node of a node n_2 on a thread t_2[2]. Because a new thread must be created as a result of the worker's move, the move includes both task switching and a thread switching, *i.e.*, creation of t_n in task switching and thread switching from t_1 to t_n

[2]Under an infinite number of processors, a new thread t_n must be always created as a result of the thread switching.

	cths	cts	ctc	cva
Protectedness		*		
Inheritedness			*	
Deterministic-locatedness				*
Deterministic-assess				*
Physical-sharedness		*		
Preservedness	*			

Table 3.3: The conditions for the constant operation: `cths`, `cts`, `ctc`, and `cva` represent respectively constant time thread switching, constant time task switching, constant time task creation, and constant time variable access.

while keeping t_1 being suspended. Provided the task switching is done by creating $gce(n_2, n_2, t_n)$, t_n will be executed without destroying $gce(n_1, n_1, t_1)$ if and only if cc provides preservedness. From (i) and (ii), thread switching is always a constant time operation, iff cc provides preservedness. □

The analysis result is summarized in Table 3.3 and 3.4.

3.5 Application of Analysis Results

In this section, we apply the analysis result in section 3.4 to a set of binding environments. In the selection of the binding environment, we exclude optimized versions of some binding environments. However, we believe that the discussion on the unoptimized ones can be mostly applied to the optimized versions of the selected subset. For example, the discussion on the Binding Array [79] can be applied to the Paged Binding Array [40].

The Binding Array (BA) [79] provides a binding array for each worker. A conditional object is defined as a word in the binding array. Attribute "owner" is `worker` because all the threads, which are scheduled in sequence with respect to a worker, use the binding array belonging to the worker. The rest of the attributes are very straightforward. The attribute-value pairs become {owner[W],association[O],instantiation[F],capacity[F]}

	Conditional Attributes				Performance Criteria				
	All	Ass	Ins	Dim	cths	cts	ctc	cva	Example
1	W	O	F	F	n	n	y	y	VA,VV,CB
2				I	n	n	y	n	
3			I	F	n	n	n	y	
4				I	n	n	n	n	
5		B	F	F	n	n	y	n	
6				I	n	n	y	n	
7			I	F	n	n	n	n	
8				I	n	n	n	n	
9	T	O	F	F	y	n	y	y	
10				I	y	n	y	y	
11			I	F	y	n	n	y	
12				I	y	n	n	n	
13		B	F	F	y	n	y	n	
14				I	y	n	y	n	
15			I	F	y	n	n	n	
16				I	y	n	n	n	
17	N	O	F	F	x	x	x	x	
18				I	y	n	y	n	TS
19			I	F	y	y	n	y	CE,RCE,DT
20				I	y	y	n	n	VI
21		B	F	F	y	y	y	n	HW
22				I	y	y	y	n	
23			I	F	y	y	n	n	
24				I	y	y	n	n	

Table 3.4: Attributes and the performance criteria

In the Version Vector (VV) [44], an array called a *version vector* is allocated for each conditional variable and its size is the same with the number of workers. According to the definition of a conditional object, a version vector itself is not defined as the conditional object. Instead, a conditional object is defined as a word in a version vector because only one cell is always accessed during the accesses of a conditional variable. It is straightforward to note that the attributes of the conditional object are the same with the Binding Array.

In the Copy-Based (CB) OR-parallel systems [5], a program is executed by a fixed number of workers in which each worker is a sequential engine with conventional four stacks and does not normally have any additional data structure for conditional objects. A conditional object is thus defined as a memory cell in the environment or the heap stack. Because such stacks are allocated for each worker, attribute "owner" is `worker`. The cells of a conditional variable are associated with the owner node, only one cell exists for each path with respect to a thread, and each cell contains only one binding. The resulting attribute-value pairs become {owner[W],association[O],instantiation[F],capacity[F]}.

According to the above discussion, the Binding Array, the Version Vector, and the Copy-Based schemes have the same attributes. It indicates that they are inherently based on the same theoretic foundation and only differ from each other in their implementations.

In the Hash Window (HW) [13], a conditional object is defined as a hash entry associated with the binding node. Each hash entry keeps only a single value that will not be updated by other threads. The attribute-value pairs become {owner[N],association[B],instance[F],capacity[F]}.

In the Variable Importation (VI) [60], a conditional object is defined as an infinite array. Given an array defined for a variable v in a node n, a cell is provided for each node in $path(n_o, n)$ if v is not yet bound; otherwise for each node in $path(n_o, b(v))$, where n_o and n_b are respectively the owner and the binding node of variable v. It should be noted that given a parent node n_p and its child node n_c the conditional object of v is defined differently unless it is not yet bound, although an infinite array is allocated for v. It is because the variable access of v in n_c needs always to access one more cell than in n_p. The array is associated with the owner node and each array is written at most once. The resulting attribute-value pairs become {owner[N],association[O],instance[I],capacity[I]}.

In the Closing Environment (CE) [24], the ROPM's Closing Environment (RCE) [53] and the Directory Tree (DT) [18], a conditional variable is renamed in each node n and a conditional object is created for the new variable. Because the owner node of the newly created variable is n, attribute "association" becomes the owner node. Each conditional object has only one cell. The attribute-value pairs thus become {owner[N],association[O],instance[I],capacity[F]}. It should be noted that these binding environments have capacity[F] whereas the Variable Importation has capacity[I]. The difference causes that the Closing Environment has a constant time variable access and the Variable Importation has a non-constant time variable access.

The Time Stamp (TS) [74] is a dynamic binding environment. As it is one of well-known binding environments, we discuss in this chapter. In the Time Stamp, a conditional object is defined as an infinite array associated with the owner node. Each cell in a conditional object is privately used by a single thread. A conditional object may have infinite cells as each of a non-deterministic number of threads may add one cell to it. For all nodes in a single path, the infinite array is always defined as their conditional object because the variable access in a node always needs to access all the cells which contain bindings regardless of the depth of the node. Therefore, the Time Stamp has instance[F] and capacity[I]. The final attribute-value pairs become {owner[N],association[O],instance[I],capacity[F]}.

With respect to the attribute-value pairs identified for each binding environment, we derived the performance criteria and listed them in Table 3.4. The performance criteria identified for the above binding environments are identical with the results obtained by Gupta [38].

3.6 Synthesis

This chapter presents a comprehensive study on the analysis of static binding environments. However, our study has a different goal from Gupta and Jarayaman's. Our study pursues a practical means which brings an insight into the design of future binding environments. Hence, it is focused on the uncovering of the entire design space of binding environments and the identification of the operational as well as the performance characteristics of each design. More specifically, it provides a framework which illustrates conceptual as well as physical aspects of the organization of binding environments and their operational aspects. We thus believe that our analysis will provide a theoretic foundation of building the more efficient binding environments.

Chapter 4

Analysis of Dynamic Binding Environments

Along with the analysis of the static binding environment, we carried out an analysis of the dynamic binding environment. As the main result of the analysis, it is proved that ideal OR-parallel logic programming implementations are theoretically possible in a system with a finite number of processors. Our analysis also shows that some semantic information must be used in the representation and management of multiply bound variables for the design of ideal OR-parallel systems. This chapter presents an analysis of the ideal OR-parallel system.

4.1 Introduction

With regard to the performance criteria, the *ideal* OR-parallel system is defined as the OR-parallel implementation of logic programs in which the variable access, task creation, and task switching are performed in a constant time not influenced by the size of the search tree and the number of variables [38]. Regarding the ideal OR-parallel system, Gupta and Jayaraman provide in a very comprehensive and theoretic analysis a proof that the design of an ideal OR-parallel system is not possible [38].

As well as to pure OR-parallel systems, the efficient design of a binding environment is still an important issue if the logic programming system exploits parallelism more extensively. This is mainly due to the fact that binding environments affect either directly or indirectly the way that other types of parallelism can be exploited [40]. This chapter revisits the issue of the ideal OR-parallel system and presents an analysis result that ideal OR-parallel systems exist theoretically. The main goal of the analysis is to benefit the parallel logic programming systems, which exploit other forms of parallelism with the OR-parallelism, as well as pure OR-parallel systems.

In this chapter, we present the analysis result mainly on the ideal OR-parallel system with a dynamic binding environment. The rest of the chapter is organized as follows. In section 4.2, a brief explanation on some theoretic concepts of task switching is offered. In section 4.3, the analysis result on one part of task switching, to be called the *deinstallation*, is presented. In section 4.4, the analysis result on the other part of task switching, to be called the *installation*, is presented. In section 4.5, the analysis is presented on the existence of the ideal OR-parallel obtained by combining the results of the previous two sections. Finally, section 4.6 summarizes the chapter.

4.2 Formal description of task switching

Assume that workers w_1 and w_2 are respectively executing threads t_1 and t_2, and w_1 attempts to perform task switching from a node n_1 on thread t_1 to a child node of node n_2 on thread t_2. Further, let the new thread to be created for worker w_1 be t_n. The task switching is denoted as $[n_1, t_1, cc_1, w_1, (n_{c_1}) \overset{n_c}{\rightrightarrows} (n_{c_2}), n_2, t_2, cc_2, w_2 \Rightarrow t_n]$, where n_c is the least (most recent) common ancestor between n_1 and n_2, n_{c_1} and n_{c_2} are the child nodes of n_c respectively in $path(n_c, n_1)$ and in $path(n_c, n_2)$, and cc_1 and cc_1 are the conditional contexts owned respectively by w_1 and w_2. During the task switching, because $gce(n_2, n_2, t_n)$ must be available for w_1 to execute t_n, the task switching is reduced to the problem of preparing $gce(n_2, n_2, t_n)$ on $gcc(n_2, n_2, t_n)$ to be the same as $gce(n_2, n_2, t_2)$. Below, the problem is describe respectively for the following cases: (I) $gcc(n_c, n_c, t_1) = gcc(n_c, n_c, t_2)$ and (II) $gcc(n_c, n_c, t_1) \neq gcc(n_c, n_c, t_2)$.

Case I: In this case, threads t_1 and t_2 share the same conditional context. The bindings made in $path(n_{c_1}, n_1)$ are never stored in $gcc(n_c, n_c, t_1)$ but always in $pcc(n_{c_1}, n_1, n_1, t_1)$. An example of this case is the Hash Window [13]. This indicates that thread t_n can share $gce(n_2, n_2, t_2)$ with thread t_2 for $gce(n_2, n_2, t_n)$. The preparation of $gce(n_2, n_2, t_n)$ is thus always a constant time operation. However, the design of an ideal OR-parallel system is not possible since the variable accesses are always non-constant time operations, regardless of whether the binding environment is static or dynamic [38, 56].

Case II: In this case, threads t_1 and t_2 do not share a conditional context, and thus t_n will have its own conditional context. An example of this case is the Binding Array. In a *static* binding environment, there are two ways to prepare $gce(n_2, n_2, t_n)$, either (a) by newly allocating $gcc(n_2, n_2, t_n)$ and then copying the values stored in $gcc(n_2, n_2, t_2)$ into it or (b) by converting $gce(n_c, n_2, t_1)$ into $gce(n_2, n_2, t_n)$. In case (a), the creation takes a non-constant time, since $gcc(n_2, n_2, t_2)$ is an infinite set[1]. In case (b), the conversion of $gce(n_c, n_2, t_1)$ into $gce(n_2, n_2, t_n)$ consists of the following parts:

1. (deinstallation:) repealing $cb(n_{c_1}, n_1)$ from $gcc(n_c, n_c, t_1)$, and
2. (installation:) installing $cb(n_{c_2}, n_2)$ into $gcc(n_2, n_2, t_1)$.

In this case, the variable access and task creation can be realized as constant time operations [38, 56]. However, since $cb(n_{c_1}, n_1)$ and $cb(n_{c_2}, n_2)$ in the above steps are infinite sets and the conversion thus always takes a non-constant time, the task switching always becomes a non-constant time operation.

Remarks: Except for a dynamic binding environment in case II, all other cases cannot be the candidates to realize an ideal OR-parallel implementation. In following sections, the ideal OR-parallel system is explored with respect to the dynamic binding environment in case II. In the analysis, the task switching and variable accesses are mainly concerned, since it is very straightforward to assess that the task creation is a constant time operation according to the findings in [38]. It should be noted that the only way to achieve the constant time task switching is to eliminated both the deinstallation and the installation parts from the task switching, while not introducing any new non-constant time operation.

[1]If the cardinality of a set cannot be limited to a fixed number, it is defined as an infinite set.

4.3 Deinstallation-Free Task Switching

In this section, it is proved that a task switching can be performed in a dynamic binding environment, in which $gcc(n_c, n_c, t_1)$ is not equal to $gcc(n_c, n_c, t_2)$, without the deinstallation part while not introducing any new constant-time operation.

4.3.1 Definitions

Associated with a thread, some definitions are explained. Note that the definitions are applied to the last path when a thread executes more than one path of the search tree.

Span: Consider a task switching $[n_1, t_1, cc_1, w_1, (n_{c_1}) \overset{n_c}{\rightarrow} (n_{c_2}), n_2, t_2, cc_2, w_2 \Rightarrow t_n]$. After the task switching, thread t_n executes $path(n_2, n_f)$. In this case, $<n_{c_2}, n_f>$ is defined as the *span* of thread t_n and is denoted as $span(t_n)$. Note that the starting node of the t_n's span is not defined as the starting node, n_2, which will be actually executed by t_n. If the thread backtracks to an ancestor node n of n_{c_2}, the starting node will also be replaced with node n.

Valid span between two subsequent threads: Consider two subsequent threads t_1 and t_2 scheduled on a conditional context[2]. The bindings made by thread t_1 are either *valid* or *invalid* for the subsequent thread t_2. The part of $span(t_1)$ that makes valid bindings is defined as the valid span of t_1 with respect to t_2 and is denoted as $vspan(t_1, t_2)$. If t_n is the last thread scheduled on a conditional context, $span(t_n)$ itself becomes the valid span and is denoted as $vspan(t_n, \_)$

Consider a thread t_n created by task switching $[n_1, t_1, cc_1, w_1, (n_{c_1}) \overset{n_c}{\rightarrow} (n_{c_2}), n_2, t_2, cc_2, w_2 \Rightarrow t_n]$. Let the spans of threads t_1 and t_n be respectively $< n_{s_1}, n_{e_1} >$ and $< n_{c_2}, n_{e_n} >$. In case of $n_{s_1} \preceq n_c$ and $n_c \preceq n_{e_n}$, the $cb(n_{s_1}, n_c)$ is valid for thread t_n, whereas the $cb(n_{c_1}, n_{e_1})$ is not valid for thread t_n. Therefore, the valid span $vspan(t_1, t_n)$ becomes $<n_{s_1}, n_c>$.

[2]In this chapter, in order to make the presentation more clear, we assume that each worker has only one conditional context. However, the discussion will be applied to the system in which a worker has multiple conditional contexts.

Subsumption, live thread, and dead thread: Consider two threads t_1 and t_2 scheduled on the same conditional context. Let t_1 be scheduled prior to t_2 and let their spans be respectively $< n_{s_1}, n_{e_1} >$ and $< n_{s_2}, n_{e_2} >$. If $n_{s_1} \succeq n_{s_2}$, then thread t_1 is *subsumed* by thread t_2. A thread t_1 is defined as a *dead* thread, if there exists at least one thread which is scheduled later than t_1 on the same conditional context and subsumes t_1. On the other hand, if t_1 is not subsumed by any thread scheduled later than t_1, thread t_1 is defined as a *live* thread.

4.3.2 Chronological partial order

For a given conditional context, the live threads are in a special relation. Consider an ordered set of threads T_{sch}=$\{t_1, t_2, \ldots, t_n\}$ scheduled consecutively by a worker with respect to the same conditional context. Let T_{live}=$\{t'_1, t'_2, \ldots, t'_m\}$ ($m \leq$n) be the set of live threads and the span of thread t'_i be $<n'_{s_i}, n'_{e_j}>$. Then a relation, to be called the *chronological partial order* relation, is always maintained among the threads in T_{live}, as described below:

- Nodes, $n'_{s_1}, n'_{s_2}, \ldots, n'_{s_m}$, have the following relation: $n'_{s_1} \preceq n'_{s_2} \cdots \preceq n'_{s_m}$.
- The valid spans, $vspan(t'_1, t'_2)$, $vspan(t'_2, t'_3)$, $\ldots$, $vspan(t'_{m_1}, t'_m)$, $vspan(t'_m, \_)$, correspond to $path(n'_{s_1}, n'_{e_m})$.

In Figure 4.1, four threads, t_1, t_2, t_3, and t_4, are scheduled for the same worker w_1 by task switching from n_1 to n_2, from n_{e_2} to n_3, and than from n_{e_3} to n_4, where n_{e_i} $(2 \leq i \leq 3)$ is the last node in the threads t_i's span. Figure 4.1 (b) depicts the span of each thread and it is noted that thread t_2 is subsumed by t_3. In this situation, T_{sch} and T_{live} are thus determined respectively as $\{t_1, t_2, t_3, t_4\}$ and $\{t_1, t_3, t_4\}$. Figure 4.1 (c) shows the chronological partial order relation among the live threads, in which the starting nodes of live threads, n_{s_1}, n_{s_3}, and n_{s_4}, appear ordered in the same path ($n_{s_1} \preceq n_{s_3} \preceq n_{s_4}$) and the valid spans of the live threads, $vspan(t_1, t_3)$, $vspan(t_3, t_4)$, $vspan(t_4, \_)$, constitute $path(n_{s_1}, n_{e_4})$.

The chronological partial order has the following two properties.

- Property 1: the valid spans of all the live threads constitutes the *current* search path, and

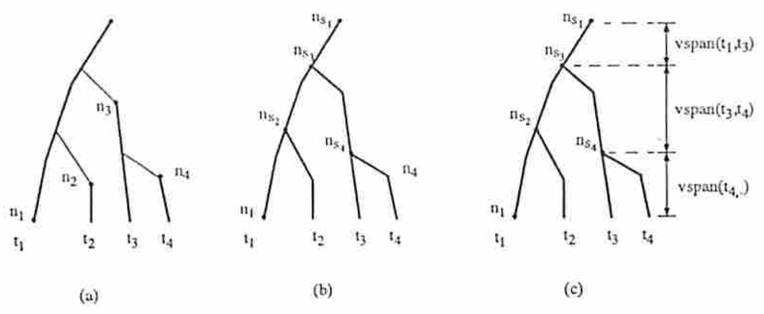


Figure 4.1: An example of the chrononical partial order relation

- Property 2: due to the single assignment property of logic variables, the bindings made in the valid spans of live threads are not destroyed during the exection of all the previous threads and are maintained in the conditional context.

The two properties indicate that the deinstallation part of task switching can be avoided if it is possible to maintain the information on live threads and their valid spans in the sytem.

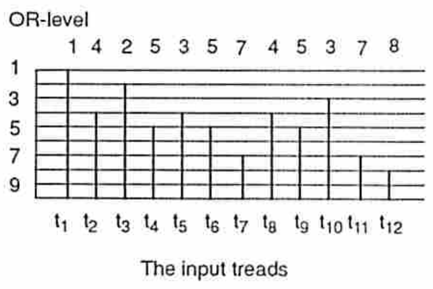
4.3.3 Description of canonical binding environment B_1

In this section, we define a canonical binding environment B_1 to explain constant time deinstallation-free task switching. In B_1, each worker w_i has a private conditional context cc_i. That is, $gcc(n, n, t_i)$ is not equal to $gcc(n, n, t_j)$ for a node n when thread t_i and t_j are executed by different workers. A conditional object of a variable is associated with its owner node. The organization and management of a conditional context are assumed the same with the Binding Array [79] except the data representation and variable dereference.

Data representation: Each entry in the conditional context has a conditional binding represented by two tuples <tag,data>. The *tag* is a value with two components <`TID`, `BOL`>, in which the `TID` is the thread identifier that made the binding and BOL is the or-level[3] of the binding node. The *data* is the conventional data term as in the WAM.

Thread table and task switching: Associated with each conditional context, a table, to be called a *thread table*, is provided. The thread table is an array in which each element contains some information on a thread. Whenever a new thread is created from a task switching with respect to a conditional context, the information on the thread is recorded in the next free element of the thread table and the offset to the thread table becomes the identifier of the thread.

Each element of the thread table has three components <LDflag, SOL,IOL>. LDflag indicates whether the thread is *live* or *dead*. SOL (Starting Or-Level) is the or-level of the starting node of the corresponding thread's span. IOL (Invalid Or-Level) is the or-level of the first invalid node after the starting node.



ADflag		D		D	D	D	D	D	D			
SOL	1	4	2	5	3	5	7	4	5	3	7	8
IOL	2		3							7	8	9
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}

Thread table

Figure 4.2: The histogram of thread spans and its thread table

Figure 4.2 depicts the histogram of thread spans for twelve threads scheduled on a conditional context, and the contents of the thread table when thread t_{12} has been executed. The blank LDflag in the thread table indicates that the thread is "live". IOL is recorded only for the "live" thread. The LDflag and the IOL field of some live threads are updated for each task switching. For example, when the task switching occurred for thread t_5, the IOL value of thread t_3 was set to 3 and the LDflag of thread t_4 was set to "D".

Under B_1, in a task switching $[n_1, t_1, cc_1, w_1, (n_{c_1}) \overset{n_c}{\Rightarrow} (n_{c_2}), n_2, t_2, cc_2, w_2 \Rightarrow t_n]$, the deinstallation of $cb(n_{c_1}, n_1)$ from cc_1 is not performed, whereas the installation

[3]The or-level of a node refers to the depth in terms of choice points created for the evaluation of ancestor nodes.

of $cb(n_{c_2}, n_2)$ into cc_2 is still performed. Instead, the update of the thread table is needed to record the newly dead threads and to adjust the valid spans of live threads affected by the task switching.

Task switching and conditional variable accesses: As the deinstallation is not performed during task switching, some bindings in a conditional context are not valid. In the dereferences of conditional variable accesses, the validity of data is checked as follows: for a binding in a conditional context, the information on the thread that made the binding is retrieved from the thread table by using the `TID` of the binding as the index to the thread table. When the LDflag indicates that the binding thread is alive, the binding is accepted as valid if the `IOL` is smaller than the BOL of the binding. In terms of the constant time condition, the variable access in B_1 is the same as in the Binding Array [79] and is a constant time operation.

4.3.4 The modified thread table

In B_1, the thread table must be updated at each task switching, whereas deinstallation is not needed in task switching. In the update, the LDflag of some live threads, which will be subsumed by the newly scheduled thread, is marked "dead", and the IOL of some live threads are adjusted. In fact, if the update is made through the inspection of each thread in the thread table, the update of thread table is not a constant time operation as the number of threads scheduled for a worker is not theoretically limited to a finite number. In other words, instead of the elimination of the deinstallation part from task switching, a new non-constant time operation is introduced. Therefore, we slightly modify B_1 to have a different form of the thread table with which the constant time update of the thread tables can be achieved.

In the modified thread table, an entry does not have the LDflag and the IOL as before. Instead, the subsumption relation regarding a group of threads are maintained in the table. The idea behind the modification is based on the fact that a group of subsequent threads scheduled on a conditional context do not have to be inspected in the update of the thread table when they show monotonously increasing values in their starting or-levels. Consider the three cases which show different patterns of the thread span histogram depicted in Figure 4.3. In all the three cases, the sequence of threads, $t_s, t_1, \ldots, t_{n-1}$, shows increasing values of their starting or-levels before a new thread t_n is scheduled, which is hereafter referred to as a triangle (in the thread span histogram) since the shape of the histogram looks like a triangle. In case (a), when t_n is scheduled, the sequence of threads including t_n still shows increasing values of the starting or-level, *i.e.* forms another triangle. As t_n does not subsume any previous thread, it is not necessary to update the thread table. In case (b), t_n has a smaller starting or-level than the previous one. It is not needed to inspect each entry of the thread table of the whole previous threads since it is enough to update the previous entry. It should also be noted that the sequence of new live threads, which does not contain t_{n-1} and includes t_n, forms another triangle, as depicted in the right-hand side of Figure 4.3 (b). In case (c), the starting or-level of thread t_n is the smaller than the previous i threads ($i \geq 2$), more precisely SOL(t_n) $\geq$ SOL(t_j) (j=s or $1 \leq j \leq n-i$) and SOL(t_n) $\leq$ SOL(t_k) ($n-i+1 \leq k \leq n$), for some i ($i \geq 2$), where SOL(t) represents the starting or-level of t. In this case, SOL(t_n) is stored in the entry of t_s in the thread table, as depicted in the small box in Figure 4.3 (c). Note that thread t_s is the first thread in the triangle formed by the previous threads and is later denoted as the owner thread of the triangle.

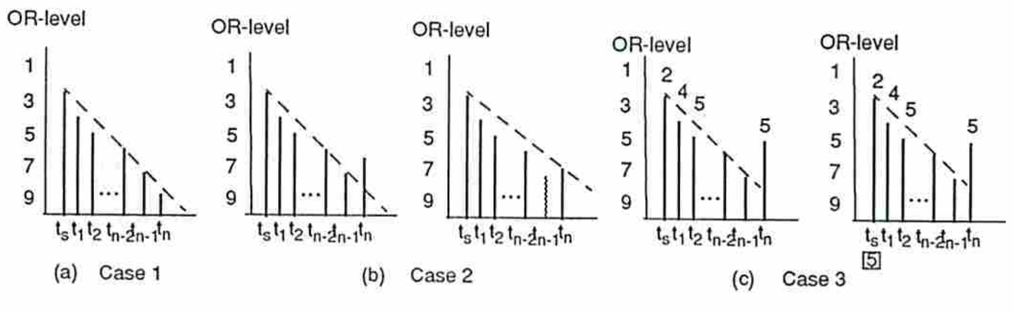


Figure 4.3: Three patterns of the thread span histogram

The triangle, a group of live threads with the increasing starting or-levels, is recursively defined as follows: a single live thread span is defined as a 0^{th}-level triangle. The n^{th}-level triangle consists of a set of triangles, where each one is an arbitrary l-level triangle pattern ($0 \leq l < n$) and at least one of them is a $(n-1)^{th}$-level triangle. Figure 4.4 shows an example represented in a shorthand notation such that a triangle is a group of threads. It shows five triangles, $T_1, T_2, \ldots, T_5$, and the first three 1^{st}-level triangles form a 2^{nd}-level triangle, and the next two forms another 2^{nd}-level triangle. Furthermore, these two 2^{nd}-level triangles form a 3^{rd}-level triangle.

The first thread of each triangle is defined as the *owner*. The starting or-level of the owner thread is defined as the *starting or-level of the triangle*. The owner thread keeps in the thread table the starting or-level of the subsequent triangle of the same triangle level. For example, the first thread of triangle T_1 becomes the owner respectively of the 1^{st}-level triangular T_1 and the 2^{nd}-level triangle formed by triangles T_1, T_2, and T_3. It thus has two values 4 and 3, the starting or-levels respectively for the subsequent 1^{st}-level triangle T_2 and for the subsequent 2^{nd}-level triangle formed by T_4 and T_5. The values stored in an owner thread are used in identifying the subsumption relation between the threads in the related triangles. Consider the triangular T_1 in Figure 4.4. At the execution of the last thread in T_5, the threads in T_1 belong to two triangles, the 1^{st}-level triangle T_1 and the 2^{nd}-level triangle formed by T_1,T_2, and T_3. In the 1^{st}-level, the threads in the area marked as region C are dead, since they are subsumed by the first thread in T_2. In the second-level, the threads in region B are dead, subsumed by the first thread of the 2^{nd}-level triangle formed by T_4 and T_5. Threads in region A remain alive since their starting or-levels are smaller than 3 (the minimum of 3 and 4). Thus, their IOLs become 3.

The modified thread table and variable access: In the access of a conditional variable, if the conditional object has a binding, we must determine (i) whether the thread which made the binding is alive or dead and (ii) whether the binding binding is valid or not if the thread is alive. For a conditional binding, the `TID` of the binding is used to find the entry of the binding thread in the thread table. Each entry in the thread table has a field that keeps pointers to each owner thread of the triangles to which the thread belongs. The `BOL` of the binding is compared with the value kept in each owner thread for every triangle level. As the result of

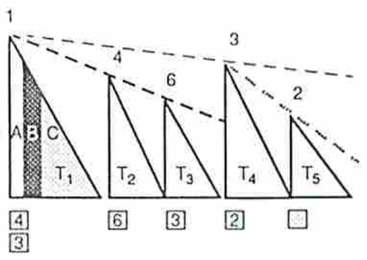


Figure 4.4: A thread span histogram with multi-triangle levels

the comparison, if the BOL is equal to or larger than any of the values, the binding is invalid since the binding thread is dead; otherwise, the binding is valid. Here, the complexity of a variable access becomes $O(L)$ since a maximum L comparisons are required, where L is the number of triangular levels created by the live threads scheduled on the conditional context.

Algorithm to update the modified thread table: The simplified version of the algorithm to update the information on some threads in a thread table is shown in Figure 4.5. The algorithm has four input parameters; `Table` is the modified thread table, `Tid` is the thread identifier of the newly created thread and is used as the index to the thread table, `input` is the starting or-level of the thread, and `Level` is the current triangular level created by the live threads. An entry of the modified thread table is represented by the following components:

1. MyOrLevel contains the starting or-node level of the thread.

2. LevelOwner is an array that keeps the identifiers for the owner thread of each triangle level.

3. NextTR contains the starting or-levels of subsequent triangles of each triangular level.

The complexity of the above algorithm is $O(L)$, where L is the number of triangle levels created by the input threads. This is the same with the conditional variable access, as explained earlier. However, it should be noted that the algorithm takes the starting or-level of the new thread. It requires that the scheduler

```
Marking(Table,Tid, Input, Level)

Begin
1:     Table[Tid].MyOrLevel := Input;
2:     for (k=1; k < Level+1; k++)
          ; Get pointer of owner nodes
3:        Table[Tid].LevelOwner[k] := Table[Tid-1].LevelOwner[k]
                .....            ; Case (b) in figure 3 is omitted
4:     if (table[Tid-2].MyOrLevel > Input) {; Case (c) in figure 3 (i=2)
5:         Table[Table[Tid-2].LevelOwner[0]].NextTR[0] = Input;
6:         Table[Table[Tid-2].LevelOwner[Level]].NextTR[Level] = Input;
7:         Table[Tid].LevelOwner[0] = Tid;
8:         Level = Level + 1;                 ; Increase triangle level
       }
End.
```

Figure 4.5: Algorithm to update the modified thread table

has to find the common ancestor node between thread t_1 and thread t_2 in a task switching from t_1 to a node in t_2. Although this will introduce a non-constant time operation in every task switching, it is at this point assumed that the scheduler can find the common ancestor node in a constant time. In order to avoid digressing from the deinstallation issue, the problem will be answered in section 4.5.

4.3.5 M-level triangular scheduling

Previously, it is shown that both the update of the thread table and variable accesses have the complexity $O(L)$, where L is the maximum value of the triangular level created by the live threads. However, if the scheduler manages program execution such that the maximum triangular level created by the live threads on each conditional context in the system is not the larger than a constant number m, the complexity will become $O(1)$. In this case, both the update of thread tables and variable accesses are reduced to constant time operations. In this section, such an OR-parallel schedule will be called an *m-level triangle scheduling.* We will provide a clue that such a schedule is feasible by proving the following proposition.

Proposition 7 *For a given logic program, there exists always an m-level triangular schedule that produces the correct answer.*

(Proof) Consider the search tree for a given logic program. Suppose that the system has only one worker w_1. Further, assume that w_1 executes the left-most path first when there exists one or more unexplored paths. In this case, when w_1 finishes a path p and $n_1, n_2, \ldots$, and n_n ($n_1 \prec n_2 \prec \ldots \prec n_n$) of the path have at least one unexplored child node, w_1 visits the nodes from n_n to n_1. At each visit it chooses the thread for the left-most child node. In the resulting thread span histogram, the first one, which corresponds to path p, is always the tallest and the height of remaining ones increase monotonously. Hence, the first thread and the last one remain alive since each thread inside the two is subsumed by the subsequent one. They thus form one-level triangular schedule. If the above principle is applied recursively to the entire search tree, the final schedule is always a one-level triangular schedule. Therefore, in a system with a single worker, there always exists an m-level triangular schedule. Suppose that an m-level triangular schedule exists each in a system with k workers, $w_1, w_2, \ldots, w_k$. Now, a new worker w_{k+1} is added to the system. It then executes the last triangle of the m-level schedule executed by w_i ($1 \leq i \leq k$) and w_i executes the remaining part except the last triangle. The other workers execute the same task which they execute in the system with k workers. In this case, both w_i and w_{k+1} have m-level triangular schedules since the schedule resulting from the removal of the last triangular from an m-level triangular schedule is still an m-level triangular schedule and also the last triangle is also a m-level triangular schedule. Therefore, all the $k+1$ workers have m-level schedules. By induction over the number of workers, it is proved that in a parallel system with more than one workers there always exists an *m-level* schedule for each worker that produces the correct answer. □

For clarity, we provide an example for which the case of a single worker is examined in Figure 4.6 (without explanation).

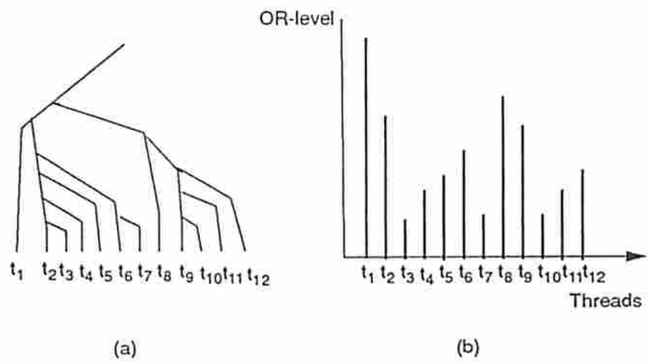

OR-level
t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12}
t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12}
Threads
(a)
(b)

Figure 4.6: An example triangular schedule

4.4 Installation-Free Task Switching

In a task switching $[n_1, t_1, cc_1, w_1, (n_{c_1}) \overset{n_c}{\rightarrow} (n_{c_2}), n_2, t_2, cc_2, w_2 \Rightarrow t_n]$ the $cb(n_{c_2}, n_2)$ needs to be installed into $gcc(n_2, n_2, t_1)$ under the binding environment with $gcc(n_c, n_c, t_1) \neq gcc(n_c, n_c, t_2)$. In this case, the installation is inherently a non-constant time operation. In this section, it is proved that the installation part of the task switching is eliminated from task switching without introducing any non-constant time operation. To this end, another canonical binding environment B_2 is defined on which the following analysis will be based.

4.4.1 Description of canonical binding environment B_2

B_2 is defined to be the same as B_1 except that instead of the thread table provided for each conditional context, one globally shared table, to be called the *common ancestor node table* (CANT), is provided. Consider two workers w_i and w_j executing respectively threads t_i and t_j with respect to conditional contexts cc_i and cc_j. Let the least common ancestor node of the two threads be n_c. Suppose that $cb(n_r, n_{cp})$ has m elements, where n_r is the root node and n_{cp} is the parent node of n_c. Then, entries $cc_i[k]$ and $cc_j[k]$ $(1 \leq k \leq m)$ B_2 are always assigned to the same variable for w_i and w_j. Hence, the bindings in the entries are always the same. The main idea behind B_2 is to maintain the information on the common ancestors between all the threads currently being executed by the workers in the system. It is used to determined whether an entry in a conditional context of a thread is for the same variable or not as to the other threads. Even if the installation is not made during task switching, we can obtain the conditional bindings from another conditional context by using the information on the common ancestors.

Common ancestor node table (CANT): The common ancestor node table is a two dimensional array. Its size is determined by the number of conditional contexts in the system. Consider a parallel system in which n workers, $w_1, w_2, \ldots,$ and w_n, cooperate to execute a logic program. CANT[i][j] $(1 \leq i, j \leq n)$, the element on i^{th} column and j^{th} row, contains the or-node level of the least common ancestor node between t_i and t_j, where t_i and t_j are the threads being executed respectively by w_i and w_j. Figure 4.7 (a) shows an example CANT in a system with four workers.

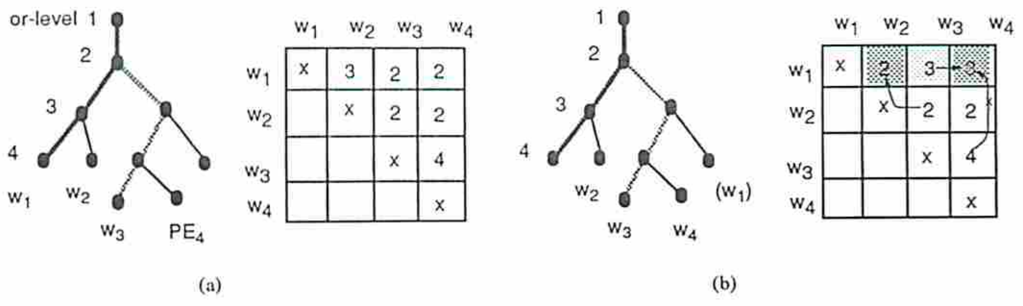


Figure 4.7: An example of the common ancestor node table

Task switching and conditional variable access: Consider an n-worker system with B_2 as its binding environment. In the system, the installation of $cb(n_{c_1}, n_1)$ is performed from $gcc(n_c, n_c, t_1)$ is normally performed in a task switching $[n_1, t_1, cc_1, w_1, (n_{c_1}) \overset{n_c}{\dashrightarrow} (n_{c_2}), n_2, t_2, cc_2, w_2 \Rightarrow t_n]$. But, the installation of $cb(n_{c_2}, n_2)$ to $gcc(n_2, n_2, t_n)$ is not performed. Instead, it is newly needed to update CANT[1][j] with the or-levels of the common ancestor nodes between the new thread t_n and thread t_j for all j $(2 \leq j \leq n)$.

In this system, some bindings concerned with worker w_i are not available in the conditional context cc_i. Suppose that a system has n workers and the k^{th} entry of conditional context cc_i is assigned to a conditional variable v. In the access of v by w_i, if cc_i[k] contains a binding, it becomes the value of v. Otherwise, w_i scans the other n-1 entries cc_j[k] $(j \neq i$ and $1 \leq j \leq n)$. If it finds a binding whose `BOL` is smaller than the CANT[i][k], it becomes the value of v; otherwise, v is an unbound variable. In this cse, the variable access becomes always a constant time operation in a system with a finite number of workers (processors).

4.4.2 Algorithm to update the CANT

Under B_2, the common ancestor node table must be updated at each task switching in order to record the or-levels of the new least common ancestor nodes between the new thread and the other threads being executed by other workers. Below, an algorithm to update the the common ancestor node table is briefly outlined.

Before a task switching $[n_1, t_1, cc_1, w_1, (n_{c_1}) \overset{n_c}{\rightarrow} (n_{c_2}), n_2, t_2, cc_2, w_2 \Rightarrow t_n]$, all the least common ancestor nodes (*lcan*) between t_2 and other threads are in the path p_{w_2} being executed by w_2. After the task switching, the *lcan* between t_n and other threads are also in path p_{w_2}. Therefore, the new entries of CANT for w_1 can be calculated directly from the entries for w_2 as follows:

- Case 1: if the or-level of the *lcan* between w_2 and w_k ($k \neq 1$, $k \neq 2$) is smaller than or equal to the or-level of n_c, it becomes the entry between w_1 and w_k;
- Case 2: if the or-level of the *lcan* between w_2 and w_k ($k \neq 1$, $k \neq 2$) is larger than the or-level of n_c, the or-level of n_c becomes the *lcan* between w_1 and w_k

Figure 4.7 shows the application of the above procedures to a task switching preformed with respect to a search tree being executed by four workers. It depicts the contents of the CANT before and after a task switching. Worker w_1 performs the task switching to the thread being executed by w_3. According to case 1, CANT[1][2] is set to 2, while CANT[1][4] becomes 3 according to case 2.

The complexity of the update procedure is $O(n)$, where n is the number of workers (processors). The update thus becomes a constant time operation in a system with a finite the number of workers.

4.4.3 Incremental schedule

Under B_2, after a task switching $[n_1, t_1, cc_1, w_1, (n_{c_1}) \overset{n_c}{\rightarrow} (n_{c_2}), n_2, t_2, cc_2, w_2 \Rightarrow t_n]$ is performed, $gcc(n_2, n_2, t_1)$ does not have $cb(n_{c_2}, n_2)$ since since $cb(n_{c_2}, n_2)$ is not installed in $gcc(n_2, n_2, t_1)$ in the task switching. Instead, w_1 uses $cb(n_{c_2}, n_2)$ in $gcc(n_2, n_2, t_2)$ of w_2. In order for w_1 to execute t_1 correctly, $cb(n_{c_2}, n_2)$ in $gcc(n_2, n_2, t_2)$ must be preserved until the evaluation of t_1 by w_1. We will analyze how the requirement of the preservation affects the scheduling, and provide a proof that it does not violate the correctness of program execution and also does not cause any non-constant time operation.

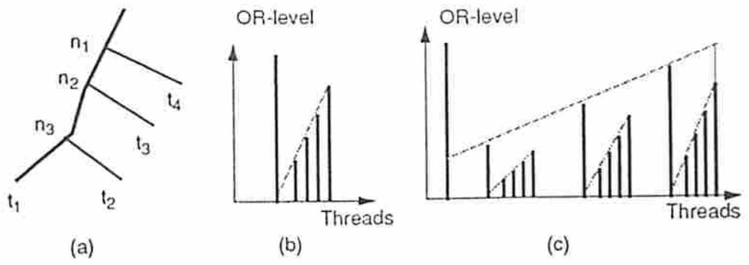


Figure 4.8: An example search tree for the incremental schedule and base patterns

Consider an example search tree depicted in Figure 4.8 (a). It contains four threads, t_1, t_2, t_3, and t_4. Suppose that w_1 has finished t_1 and is considering task switching to another unexplored thread. Let us consider the following two cases:

- Case (a): thread t_4 is being executed by a worker other than w_i.
- Case (b): thread t_4 has not been executed and w_i selects thread t_4.

In case (a), w_1 must not perform task switching to a node n ($n \preceq n_1$). Otherwise, bindings $cb(n, n_1)$ on $gcc(n_1, n_1, t_1)$ will be destroyed, and then thread t_4 will not find some bindings in $cb(n, n_1)$ when it tries to access them. In case (b), the bindings $cb(n_1, n_3)$ will be destroyed. Later, when any worker including w_1 attempts to execute t_2 or t_3, it cannot find the bindings and fails to correctly execute the threads.

The above discussion indicates that the scheduling must be performed such that any shared bindings (bindings to be used by other threads) are not destroyed. To this end, we examine a special scheduling strategy which we will call an *incremental schedule.* The main idea of the incremental schedule is that each worker executes the bottom-most one among available threads. Under the incremental schedule, each worker has a thread span histogram which shows a specific pattern, called an *incremental pattern.* An incremental pattern is defined recursively as follows. A single thread span is defined as a 0^{th}-level incremental patten. The n^{th}-level incremental pattern consists of a set of other incremental patterns. The first one is always a 0^{th}-level incremental pattern showing the highest span. The remaining incremental patterns are ordered by increasing height, where each one is an arbitrary l-level incremental pattern ($0 \leq m < n$) and at least one of them is a $(n-1)^{th}$ incremental pattern. Figure 4.8 (b) shows a 1^{th}-level incremental patten. When a schedule results in an $(m)^{th}$-level incremental patten, it is called an *m-level incremental schedule.* Figure 4.8 (c) shows a 2-level incremental schedule, in which a 0^{th}-level incremental pattern and three 1^{st}-level incremental pattern makes a 2^{th}-level incremental pattern.

With respect to the incremental schedule, the problem is to determine whether there always exists an incremental schedule for a logic program. We answer the problem by proving the following proposition.

Proposition 8 *For a given logic program, there always exist an incremental schedule that produces the correct answer in a parallel system with an arbitrary number of workers.*

(Proof) In the proof of Proposition 1, we introduced a specific scheduling strategy in order to prove that for a system with one worker, there always exists an m-level triangular schedule. Suppose that the same strategy for the case of one worker is applied. In the proof, it is shown that in terms of thread spans, the first one, which corresponds to path p, is always the tallest and the height of remaining ones increase monotonously. According to the definition of the context of the increment schedule, it is clear that the resulting thread span histogram is exactly an incremental schedule. Therefore, it is proved that for one worker, there always exists an incremental schedule. As the other part of the proof is very straightforward with almost the same induction used in Proposition 1, we do not expose the exact proof procedure in this chapter. $\square$

The example in Figure 4.6 is also used to examine the proof procedure.

4.5 Constant Time Task Switching

In the previous sections, it has been found that under B_1, a task switching is performed without the deinstallation and under B_2, without the installation. Neither case introduces any non-constant time operation, provided the m-level triangular schedule for the former and the incremental schedule for the latter are ensured in a system with a finite number of workers (processors). This section presents the analysis result regarding the task switching to be performed without both the deinstallation and installation at the same time.

First, a new binding environment B_3 is defined that combines B_1 and B_2. Because the semantic information maintained in the thread tables of B_1 does not conflict with the information in the common ancestor table of B_2, it is always possible to combine them. However, the problem with the unified binding environment B_3 is to determine whether both the m-level triangular schedule and the incremental schedule are satisfied at the same time. The problem is answered by the following two propositions.

Proposition 9 *For a given search tree, an incremental schedule on a conditional context is always reduced to an m-level triangular scheduler.*

(Proof) As discussed earlier, the thread span histogram of an incremental schedule is composed of an n^{th}-level incremental patten. (i) First, a 0-level incremental schedule is always a triangle since both are defined as a single thread span. (ii) Now, consider an n-level incremental schedule. It consists of a 0^{th}-level incremental pattern ts_n and a set of incremental patterns, $ip_1, ip_2, \ldots, ip_m$. In the incremental pattern, the first thread span ts_n and the last one ip_m remain as the live threads since the or-level of $ip_{(i-1)}$ is larger than ip_i $(2 \leq i \leq m)$, *i.e.*, $ip_{(i-1)}$ is subsumed by ip_i. and the or-level of ts_n is smaller than ip_m. The ip_m results in two live threads, the first one $ts_{(n-1)}$ and the last incremental pattern. Together with ts_n, they form a triangular. If the reduction process is applied recursively, the final thread span histogram obtained is reduced to a triangle. From (i) and (ii), an incremental schedule is always reduced to a 1-level triangle. □

Proposition 10 *Under B_3, for a given logic program, there always exists a schedule which violate neither the m-level schedule nor the incremental schedule in a system with a finite number of workers.*

(Proof) Under B_3, there always exist an incremental schedule according to Proposition 2 since B_3 is the superset of B_2, The incremental schedule is always reduced to an m-level triangular schedule according to Proposition 3, where m is 1. □

According to Proposition 3, the task switching can be performed in B_3 without both the deinstallation and installation at the same time, while not introducing any new non-constant time operation. In section 4.3, it is noted that the common ancestor node n_c needs be found in every task switching $[n_1, t_1, cc_1, w_1, (n_{c_1}) \overset{n_c}{\Rightarrow} (n_{c_2}), n_2, t_2, cc_2, w_2 \Rightarrow t_n]$. It is not a constant time operation in B_1, as opposed to in B_2. However, in B_3, it does not cause any non-constant time operations, since the scheduler can now use the CANT discussed in section 4.4. Consequently, in B_3, the task switching and variable accesses are constant time operations in a system with a finite number of processors. The task creation is inherently a constant time in a binding environment like B_3, as was proved in [38]. In consequence, it is concluded that the ideal OR-parallel logic programming system exists theoretically.

4.6 Synthesis

This chapter has shown a proof that ideal OR-parallel logic programming implementations are theoretically possible in a system with a finite number of processors. Our analysis has shown that an ideal OR-parallel system was possible *only* when some semantic information is used in the representation and management of conditional variables. Indeed, in the analysis, the inference depth of a binding node and a thread identifier are used in the data representation. We believe that our finding will provide a theoretic foundation for more efficient parallel logic programming systems.

Chapter 5

The Parallel Execution Model

In the previous chapter, we have found that the ideal OR-parallel execution systems exist in the theoretic design space. According to the result, the ideal OR-parallel systems are possible only when some semantic information is used in the representation of conditional bindings. We have applied the findings to the design of an OR-parallel execution model. The main objective of the design is to develop a parallel execution model which is not only highly flexible to make runtime schedulers be able to implement a variety of scheduling algorithms and architectural optimizations but also is highly efficient to make runtime schedulers be able to obtain the expected performance from the implementation of such algorithms. This section presents the design of our parallel execution model.

5.1 Introduction

In the presence of parallelism, one of important issues is the allocation of parallelism. The allocation can be made before execution, provided the parallelism can be identified at compile time. Usually, this will turn out more efficient than doing the allocation at runtime. Unfortunately, the parallelism in logic programs is very difficult to analyze at compile time. Indeed, almost every parallel logic system relies on dynamic allocation made by *runtime schedulers*. The runtime schedulers play an important role in the parallel execution of logic programs and greatly affect the parallel performance. Therefore, the design of an efficient scheduler is one of the most important issues in building parallel logic programming systems.

In the implementation of parallel logic systems, the design of the low level software is tightly coupled with a number of system components. The runtime behavior of logic programs has been frequently represented by search trees [5, 15]. These search trees describe in a conceptual level the behavior of program execution. In this respect, the program execution is viewed as a parallel traversal of the search tree. In an OR-parallel system, when each worker executes a node, the global environment of the node must be provided for the worker as discussed in the previous chapters. To this end, a specific memory model is employed in an OR-parallel system. The memory model determines the cost of task switching and variable accesses. As task switchings are originated from the scheduling in a parallel logic programming system, the scheduling framework is strongly coupled with the memory model employed for the management of multiple bindings. In this respect, in the design of a runtime scheduler, the interaction between the memory model and the scheduler must be thoroughly investigated in order to obtained high system performance.

In parallel logic systems, the design of low level system software has additional dimensions of complexity. In the parallel execution of conventional programs, the high system utilization has been one of the major performance criteria. This is because higher utilization normally results in higher performance. In this respect, the system state is assessed as either *busy* or *idle* and the design of low level software is focused on making each processor as busy as possible. In the parallel execution of logic programs, high system utilization does not necessarily present high performance. This is because logic programs rely heavily on speculative computation. For example, almost all the PROLOG programs use "cut" for reasons of efficiency. Its main function is pruning some portion of computation and thus making it unnecessary in the execution. The speculative computation is caused by "cut" because before the execution of "cut", it is not possible to determine whether a given portion of computation will be pruned away or not. That is, in order to avoid losing the parallelism, if the system will execute a portion before it is found out that the portion will not be pruned, the system may waist system resources for unnecessary computation. In this respect, logic programming system have one additional state of executing unnecessary work and in the design of a runtime scheduler, a special concern must be made with regard to the speculative aspect of logic programs.

On the architectural side, the design of low level software must consider the architectural specification. For example, in large-scale parallel machines, the latency of memory accesses changes according to the physical distance between processors. In such systems, an architectural optimization which reduces the total amount of remote accesses is crucial for performance enhancement. In this respect, the design of the runtime scheduler must be highly flexible with regard to architectural optimizations and provide some support which makes such optimizations efficient.

In the above discussion, we defined within the context of logic programming systems three important requirements for the design of low level software as follows.

1. The efficient interaction must be achieved with the low level components in the execution model
2. The speculative aspects of logic programs must be considered.
3. The architectural specification must be optimized.

In order to achieve the expected performance, the parallel execution model must satisfy the above requirements. This is accomplished by providing some parallel support which help the low level software in general and the runtime scheduler in particular to efficiently achieve their functions in their implementation. Upon the requirements, we have designed a parallel execution model. Our parallel execution model satisfies the first requirement by providing the Tagged Binding Array. In order to address the other two requirements particularly for the design of the runtime scheduler, our execution model provides a parallel support which we will called *least common ancestor based scheduling support.* Overall, assessed in terms of the performance criteria, our execution model has the following characteristics:

1. high single thread performance, and
2. low scheduling cost to adapt highly irregular shapes of the search tree created in real world application programs.

Other than the Tagged Binding Array and the common ancestor tree based scheduling support, as the other important components, our execution model provides the flat indexing scheme and a translation based execution mechanism of Prolog programs. Below, we introduce them briefly and the detailed discussion will be offered in the following chapters.

The flat indexing scheme

To find an optimal indexing is quite complex and also demands a large number of abstract machine instructions for its implementation. In the indexing scheme of the WAM, the first argument of a clause is used as a key. Viewed from the trade-off between the efficiency and simplicity, the usage of the first argument as a key for indexing is a quite reasonable choice. However, viewed from OR-parallel execution, the indexing scheme of the WAM is inefficient in terms of both parallelism and scheduling efficiency. The main reason for this is that an invocation of some goal may result in the creation of more than one choice points. After investigating the problem in detail, we designed a new indexing scheme, which we will call *flat indexing*. Under the flat indexing scheme, the maximum number of choice points created for an invocation of a goal is always one. Therefore, it makes parallelism be exposed earlier than the indexing scheme of the WAM. As a result, the degree of parallelism of each node becomes bigger. The details will be discussed separately in chapter 7.

Translated execution of PROLOG prolog programs

In terms of execution speed, compiling PROLOG programs into low-level assembly language code is the most efficient way. Due to low portability, this approach is very expensive when the prototype is re-targeted to other platforms. To enhance the portability, two other approaches have been explored elsewhere. One consists in an emulation of the WAM (Warren Abstract Machine) code in C-code and the other is based on the translation of PROLOG programs into C language programs. In spite of its simplicity, the emulation approach is less advantageous since it is quite slower than the translation approach. However, the translation approach is difficult due to some problems. For example, the translation of the flat structure of PROLOG programs into the function calls in the C language causes much inefficiency in terms of execution speed. To solve those problems, we developed an efficient Prolog to C translation method. We then applied it to the design of the sequential engine of our parallel logic programming system. The details will be discussed separately in chapter 6.

Behind the design of a new parallel execution model, we selected the OR-parallelism as the main source of parallelism to exploit. The reason for this choice is that the efficient exploitation of the OR-parallel is important for the other types of parallelism such as AND-parallelism. This is particularly true when the execution chooses the binding environment which has been opted for the high single single thread performance over low cost scheduling [5, 15].

The rest of the chapter is organized as follows. Section 5.2 presents the methodology to manage the multiple environments of our execution model. Section 5.2.3 presents the components of our execution model which supports runtime scheduling. Section 5.4 summarizes the chapter.

5.2 The Tagged Binding Environment

As one of important components in our parallel execution model, we have developed the Tagged Binding Environment. The key objective of the design is to reduce the cost of task scheduling by making a task switching efficient. This section presents the Tagged Binding Environment.

5.2.1 Motivation

The binding environment has been a highly important issues in the parallel implementation of logic programs. In the past few years, many schemes for the binding environment has been proposed [18, 7, 25, 40, 81, 82] for efficient OR-parallel execution of logic programs. However, Gupta shows that the binding environment has also crucial impacts on the efficient exploitation of the other forms of parallelism[40]. The binding environment has thus increasing importance as the logic programming systems exploits the parallelism more extensively. In spite of high efficiency of existing binding environments in the implementation of logic languages on small-scale parallel machines, the performance scalability of the schemes on large-scale machines is not clear; some binding environments such as the Binding Array cause very high overhead in task scheduling, while the closed binding environment method causes very poor single thread performance due to intolerable parallel overhead and inability to control scheduling. Therefore, along with advanced research in the high level, the research on the binding method is essential for expected scale-up in performance on large-scale parallel architectures.

The organization of a binding environment determines the single thread performance and the cost of scheduling. In logic programming languages, the ideal single thread performance and ideal low cost task scheduling cannot be obtained simultaneous by one binding environment. Hence, one of them is sacrificed in the design of the binding environments. In the design of the popular binding environments, the single thread performance is chosen over the low cost scheduling. The decision is based on the argument that for a given binding environment, single thread performance, determined mainly by the cost of the variable accesses, cannot be optimized since the amount of variable accesses is program dependent, but the amount of scheduling can be minimized by the highly advanced scheduler. Within the context of the small-scale multiprocessors, this has been a reasonable choice since the requirement of scheduling is usually small. But, it is observed that the requirement of task switching in scheduling increases more rapidly as the system becomes larger-scale. Therefore, the issues of task scheduling need to be more highly considered for the design of the binding environments in the large-scale parallel machines.

In order to access the issues of task scheduling pertaining to the design of binding environment, we have performed a comprehensive analysis of binding environments as presented in chapter 3 and 4. As a result, in the design of a multiple binding method suitable for the large-scale parallel architectures, we set the following two goals:

1. high single thread performance, and
2. low scheduling cost to adapt highly irregular shapes of the search tree on real world application programs.

To achieve the goals, we applied the findings obtained from the analysis of the ideal OR-parallel systems. The binding environment thus developed is the Tagged Binding Array.

The rest of the section is organized as follows. Subsection 5.2.2 presents a review of the Binding Array method to provide a frame for the discussion of the proposed binding environment. Subsection 5.2.3 presents the Tagged Binding Array.

5.2.2 Review of the Binding Array

The Binding Array was proposed by D.S. Warren [79]. The Binding Array was subsequently adopted in the Gigalips project as the SRI model [81] and in the implementation of the parallel logic language BRAVE [70].

Memory organization

The Binding Array uses two auxiliary data structures. the *forward list* assigned to each node and the *binding array* assigned to each PE. The forward list of a node is an ordered list which keeps the conditional bindings made in the node. It is the modified version of the trail stack in the conventional sequential WAM. Different from the trail stack in which each entry contains only the variable address, each entry of the forward list is represented by <variable address, binding>. The forward lists of the search tree serve as the globally shared binding tree. The forward lists is used to repeal some bindings at backtracking and to restore some bindings in at scheduling.

The binding array of a PE is an order list which keeps all the conditional bindings made in the search path being computed by the PE. It assumes the role of a cache for conditional bindings. Each cell of the binding array is assigned to a conditional variable and keeps its binding.

Variable binding and dereferencing

Associated with each node of the search tree, a counter is provided to manage the binding array. Initialized to 0 at the root node, the parent node' counter is copied into its children nodes whenever a branch takes place. When a conditional variable is created in a branch, the value of the counter is stored in the environment frame slot which is assigned to the variable and the counter is incremented. The value serves as the name of the binding array cell assigned to the variable.

The reference of variable bindings is supported via a link between the environment frame slot and a binding array cell. When a binding is made for a conditional variable on a PE, the binding is written into a binding array cell which is assigned to the variable. The index stored in the environment frame slot is used to address the binding array cell. In order to access the binding, we read the index from the environment frame cell and then retrieve the data from the binding array cell. The precise algorithm is described in Figure 5.2.2.

```
algorithm    dereference(V)
input
          V                                       : input term
begin
          if ( V.tag == Var ) {
                    if ( V.value == V )
                              return V;
                    else
                              return dereference(V);
          }
          else {
                    if ( V.tag != NON-VAR );
                              return V;
                    else {
                              Bdata = BA[i];
                              if ( Bdata.value != value )
                                        return dereference(Bdata)
                              else
                                        return Bdata;
                    }
          }
end
```

Figure 5.1: Dereference algorithm in the Binding Array

Compared with other binding methods, the Binding Array provides very cheap variable accesses. Indeed, a variable access includes just one additional level of indirection, compared with the one in the sequential WAM. Moreover, the management of the binding array is very simple because a binding array is organized as a stack. The efficiency of variable accesses and the simplicity of binding array management originate from the property that the contents of each binding array reflect the environment of a path. However, this causes a disadvantage that the cost of task switchings is very high. When a worker switches its task from one node to another, the binding array must be adjusted to make the contents reflect the environment of the destination node. The update consists of two steps: (1) the deinstallation of bindings made between the current node and the least common ancestor node and (2) the installation of bindings made between the common ancestor node and the destination node. As both the deinstallation and the installation must be performed in sequential, task switching becomes very expensive.

5.2.3 Tagged Binding Array (TBA)

This section presents its motivation, memory organization, and operations of the Tagged Binding Array.

Overview of the Tagged Binding Array

Figure 5.2 depicts the relation between a task switching and the contents of the environment. In the figure, a task switching occurs for a worker w_1 from a node n_4 to another node n_5 located in a different path being executed by worker w_2. Before the task switching, the global environment of the least common ancestor node n_3 between w_1 and w_2 contains the bindings made both in $path(n_1, n3)$ and in $path(n_3, n_4)$. It is depicted in Figure 5.2 (a).

In a normal task switching, the bindings made in $path(n_3, n_4)$ are removed (deinstalled) from the environment of w_1 and also the bindings made in $path(n_3, n_5)$ are imported (installed) into the environment. In consequence, the environment keeps only the bindings made in $path(n_1, n_3)$ and $path(n_3, n_5)$, as depicted in Figure 5.2 (b).

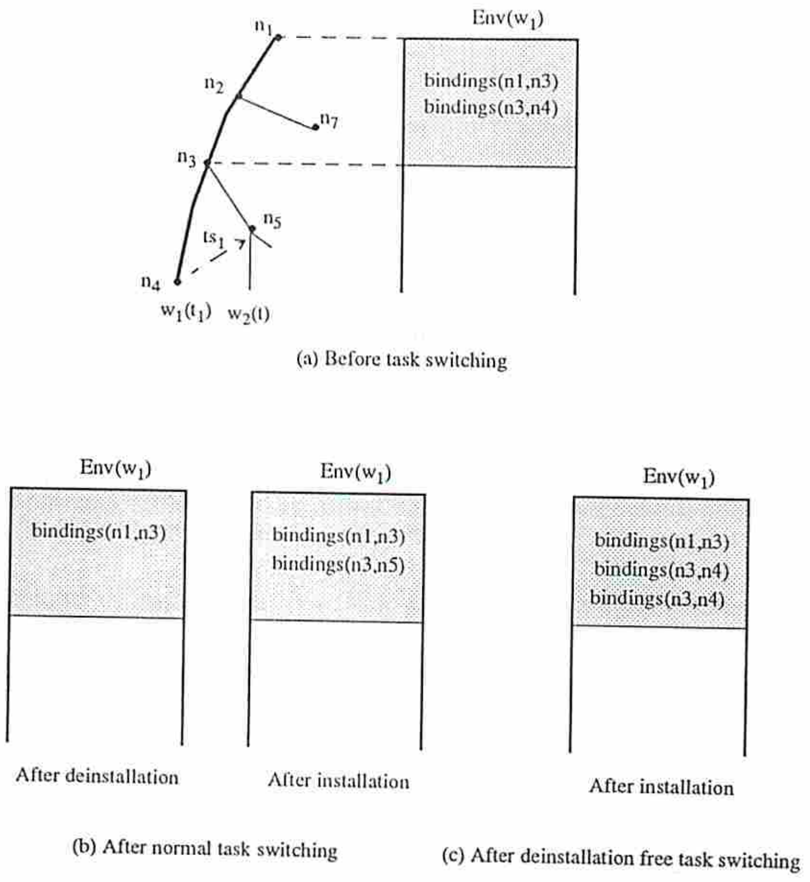


Figure 5.2: Task switching and environment preparation

In order to reduce the cost of task switching, we developed a new binding environment which we will call *Tagged Binding Array.* The Tagged Binding Array is aimed at reducing the cost of task switching, while preserving the constant time variable accesses. In order to preserve the constant time variable accesses, it uses a data structure which is similar to a binding array in the Binding Array. To realize cheaper task switchings, it removes the deinstallation part from task switchings. As a matter of fact, this principle originates from the concept of the deinstallation-free task switching explained in chapter 4. The reason for not removing the installation part is that without specific hardware support such as associated memory, the removal of the installation part would rather cause harmful influence on the system performance due to the newly introduced overhead.

If the deinstallation step is omitted, the contents of the environment will be different from the one resulting from a normal task switching. That is, the environment contains the bindings made in $path(n_3, n_4)$ as well as those resulting from the normal task switching. This is depicted in Figure 5.2 (c). In this case, the bindings made in $path(n_3, n_4)$ must be ignored in variable accesses because they are not the part of the environment. In the next subsection, we present the design of the Tagged Binding Array and illustrate how such bindings are handled.

Organization of the Tagged Binding Array

In the previous subsection, it was noted that the principle of the Tagged Binding Array originates from the concept of the deinstallation-free task switching. In other words, the Tagged Binding Array is the implementation model of the canonical binding environment B_1 described in chapter 4. However, the Tagged Binding Array is quite different because the conceptual representation of B_1 is transformed to an efficient form in its implementation. As a main difference, the thread table provided for with each binding array (conditional context) is replaced with the following two data structure: the *tagged binding array* and the *thread span stack.*

Tagged binding array: Except the data representation and its variable dereferences, a binding array in the Tagged Binding Array, which we will call *tagged binding array*, is exactly the same with the Binding Array in terms of its role and operations.

In its data representation, each entry in the tagged binding array has a conditional binding represented by two tuples <stag,data>. The *stag* is a value with two components `<tid, bol>`, in which `tid` is the name of the thread that made the binding and `bol` is the OR-level[1] of the binding node. The *data* is the conventional data term of the WAM.

[1]The OR-level of a node refers to the depth in terms of choice points created for the evaluation of ancestor nodes.

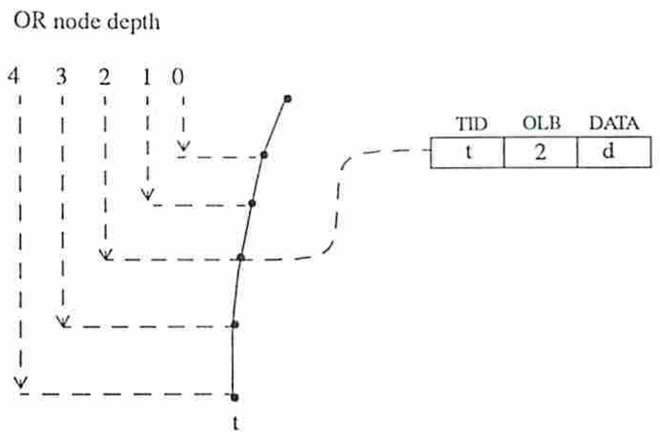


Figure 5.3: Data representation in the TBA

Figure 5.3 shows a thread which executes a path. It depicts the OR-level assigned to each node and a binding made by the node with its OR-level as "2". The binding is represented by <t,2,d>, where t is the name of the thread, 2 is the OR-level of the binding node, and d is the data term.

Task span stack and the TSP register: In chapter 4, we discussed that a thread table is provided with respect to each worker to maintain the valid span of threads. At each task switching, the thread table must be adjusted for all the previous threads whose valid spans are influenced by the task switching. In terms of the implementation, the update of the thread table is somewhat expensive even though it can be performed in a constant time. Therefore, we introduce the thread span stack in order make identification of valid spans simple and efficient.

Like the thread table, the thread span stack is based on the chronological partial order relation discussed in chapter 4. The chronological partial order is maintained among the live threads. Let T_{live}=$\{t'_1, t'_2, \ldots, t'_m\}$ ($m \leq$n) be the set of live threads and the span of thread t'_i be $<n'_{s_i}, n'_{e_j}>$. The chronological partial order relation is summarized as follows:

- The nodes, $n'_{s_1}, n'_{s_2}, \ldots, n'_{s_m}$ have the following relation: $n'_{s_1} \preceq n'_{s_2} \cdots \preceq n'_{s_m}$.
- The valid spans, $vspan(t'_1, t'_2)$, $vspan(t'_2, t'_3)$, ..., $vspan(t'_{m_1}, t'_m)$, $vspan(t'_m, \text{-})$, correspond to $path(n'_{s_1}, n'_{e_m})$.

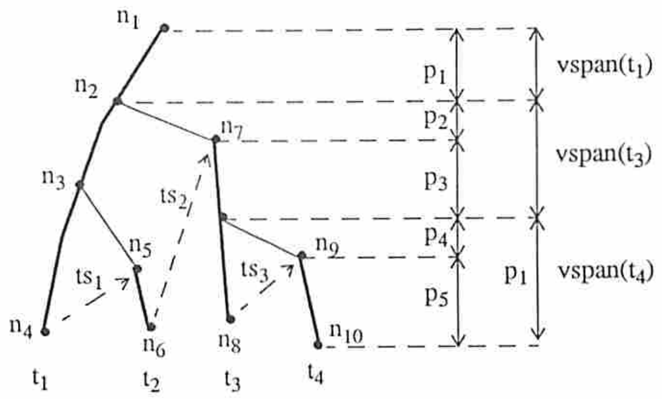


Figure 5.4: An example vspan

Figure 5.4 shows an example chronological partial order when the worker is executing thread t_4. In the example, before thread t_4, three task switchings, *ts1, ts2*, and *ts3*, occurred in sequence with respect to threads, t_1, t_2, and t_3. In consequence of the task switching, the three threads t_1, t_3, and t_4 constitute the path. The figure shows the valid spans for the path.

In chapter 3, we used a thread table to maintain the information on the valid span of each thread; the thread table provides an entry for each thread. The drawback of this approach is that it is not efficient to update the table for each task switching because we need to inspect each entry of the table. In order to avoid the drawback, we can use a different representation of the valid spans. Whereas in the thread table, the information on a valid span is maintained for each thread, we can keep the information on the valid span in each node to accomplish a more compact and efficient representation. This is possible because according to the chronological partial order, a node in a path always belongs to a single valid span of a thread.

The implementation thus obtained is the thread span stack. A thread span stack provides an entry for each node in the current path such that all the entries are ordered by the OR-node depth. Each entry of the thread span stack is the name of the thread whose valid span the node belongs to. The TSP register has a pointer to the current top of the thread span stack. When the worker is executing a branch of the n^{th} node from the root, the thread span stack contains n entries and the TSP register points to the n^{th} element. It should be noted the the thread table and the thread span stack use respectively the thread and the node as a carrier of the valid span. In the thread table, each thread keeps the information of its valid span via nodes, while in the thread span stack each node keeps the name of thread whose valid span it belongs to.

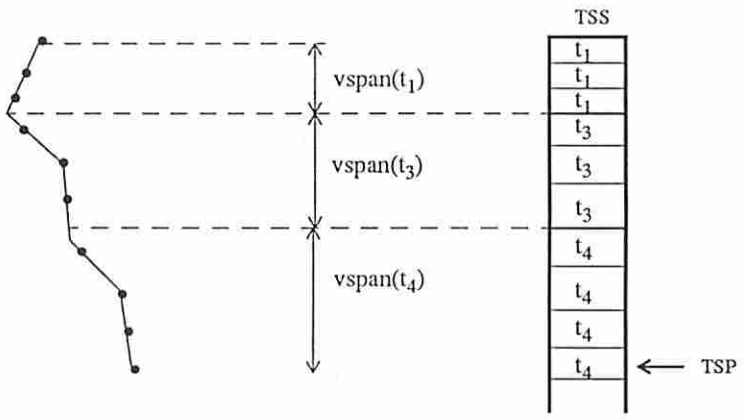


Figure 5.5: Thread span stack

Figure 5.5 shows an example valid span and the thread span stack. In the figure, it is shown that the first three entries of the TSS are filled with t_1, the next four entries with t_2, and finally the next 3 entries with t_3, and the TSP register points to the last element of the TSS.

The relation between the tagged binding array and the thread span stack: If a binding was made by any thread other than the live threads, it is not valid. Hence, data values in the tagged binding environment have either one of the two states: valid or invalid. In order to check if a binding in a tagged binding array is valid or not, we use the the thread span as follows.

Consider a binding D represented in $<D.tid,D.bol,value>$, where $D.tid$ is the thread which made the binding, $D.bol$ is the binding OR-level, and $value$ is the data term. Using the thread span stack, we can identify live threads which constitute the current valid spans. If $D.bol$ is less than or equal to the value of the TSP register, the node in the $D.bol^{th}$ depth is available in the current path; otherwise, the binding is by default not valid. The content of TSS[$D.bol$] is a thread t_v to which the node in the $D.bol^{th}$ depth of the valid span belongs. If binding thread $D.tid$ is different from t_v, binding D is invalid because it is made by a thread which is not alive.

Operational principle of the Tagged Binding Array

This subsection presents the management of the thread span stack, dereference mechanism, and finally the task switching.

Management of the tagged binding array: As we pointed out earlier, the management of the thread span stack is very simple since the stack reflects exactly a search path with its entries. The only operations pertaining to the management are normal "push" and "pop", and the TSP register serves as the stack pointer. These are summarized as follows:

- At each creation of a choice point, the name of the current thread is pushed onto the thread span stack.
- At each backtracking or task switching, the top of stack is popped out.

Variable dereference: In the Tagged Binding Array, the procedure to dereference a variable is an extension of the one in the Binding Array. The extension is made for checking the validity of a binding. Once a binding is found out valid, the rest of the procedure is the same with the one in the Binding Array. Otherwise, the dereferenced variable is regarded as an unbound variable. As an optimization, we remove the invalid binding from the tagged binding array in order to avoid the validity check in the subsequent dereferences. The complete dereference algorithm is depicted in Figure 5.6.

To make the dereference procedure clear, we provide an example in Figure 5.7. The example shows three bindings in a tagged binding array. According to the contents of the binding array, the binding in the first entry was made by t_3

```
algorithm    dereference(V)
input
    V                                        : input term
begin
    if ( V.tag == Var ) {
        if ( V.value != V )
            return V;
        else
            return dereference(V);
    }
    else {
        if ( V.tag != NON-VAR );
            return V;
        else {
            Bdata = BA[i];
            if ( Bdata.value == value )
                return Bdata;
            if ( V.bol ≤ TSP and  TSP[V.bol] == V.tid )
                return dereference(V);
            else {
                V.value = V;
                return V.value;
            }
        }
    }
end
```

Figure 5.6: Dereference algorithm in the Tagged Binding Array

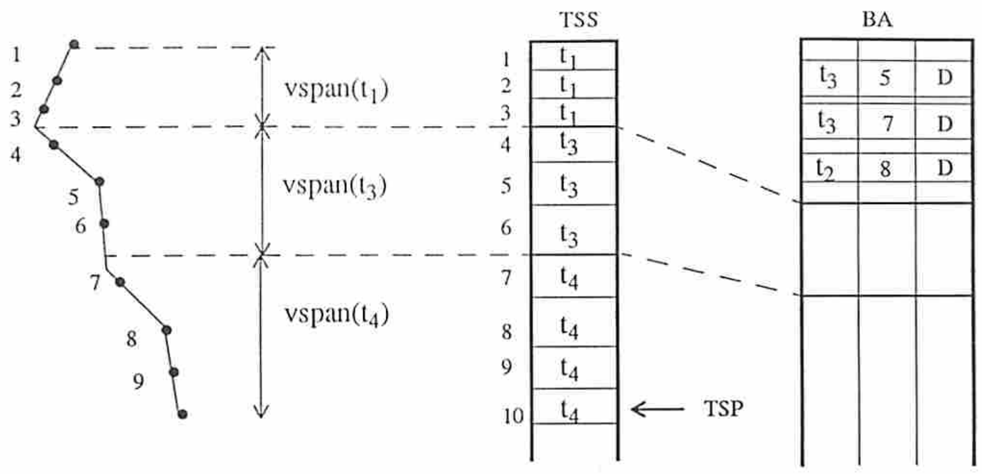


Figure 5.7: An example dereference

at OR-level 5. The binding in the second entry was made by t_3 at OR-level 7. The binding in the third was made by t_2 at OR-level 8. Among them, only the first one is valid because it is made within the current valid span by a live thread. Even though the second one was made by a currently live thread, it is not valid because the binding was made out of the valid span of the thread. The third one is not valid because it wad made by the thread which does not belong to the set of live threads.

Task switching: As discussed earlier, a task switching in the Tagged Binding Array is performed without the deinstallation part. Consider a task switch from a node n_s to a node n_d which occurs with respect to a worker w_i. Let the least common ancestor node be n_c and let the OR-node levels of n_s, n_d and n_c be Ol_s, Ol_c, and Ol_d, respectively. Let us define the new thread in the worker be t_n. The task switching consists of the following major parts: the installation of bindings and the update of the thread span stack and the TSP register, as described below.

- Install the binding made between nodes n_c and n_d in the trail stack. (Note that the thread name of the binding is changed to "t_n".)
- Fill TSS[Ol_c+1] to TSS[Ol_d] with t_n.
- Set the TSP register to Ol_d.

In the above steps, it is noted that each binding has a new thread as its binding thread. Task switching always causes the creation of a new thread. The installation is inherently an importation of variable bindings from another worker. When a binding is installed, although the binding OR-level is still effective for the new worker, the binding thread cannot be recognized by the new worker. To solve this problem, it is viewed that the bindings to be installed are conceptually imported from the original binding thread into the new thread. Therefore, the bindings to be installed into w_i are regarded as those made by t_n and thus their `tid` field is set to t_n during the installation.

5.3 The Least Common Ancestor (LCA) based Scheduling Support

The parallelism in logic programs is very difficult to analyze at compile time. Indeed, almost every parallel logic program system depends on dynamic allocation by *runtime schedulers*. The schedulers play an important role in the parallel execution of logic programs and greatly affect the parallel performance. Therefore, the design of an efficient scheduler is one of the most important issues in building parallel logic programming systems.

In order to get efficient implementations of low level system software in general and the runtime scheduler in particular, the parallel execution model must provide appropriate parallel supports. This supports ranges from low level components such as the binding environment to high level execution principle such as efficient runtime search tree representation. Our execution model provides a parallel support which we will called *least common ancestor based scheduling support.* The rest of the section presents the parallel support in detail.

5.3.1 An analytic model of the tree-based scheduling

In the previous chapter, we showed the runtime behavior of a Prolog program in the form of a OR-parallel search tree. The OR-parallel search tree clearly represents the execution behavior and provides a sound basis for the scheduling.

Most parallel logic programming systems which take the approach to thread-based execution exploit the WAM as their sequential engine. In such systems, the search tree is implicitly constructed by means of the internal data structure such as choice points. In the search tree, the OR-parallel tasks are represented as the unexplored branches and the scheduling is described as the assignment of a processor to a node which has some unexplored branches. Scheduling in such systems is in principal performed with information associated with the search tree. Such schedulers is frequently called *tree-based* schedulers.

Some parallel logic programming systems take the approach to process-based execution. In such systems, a program is executed in the absence of the globally shared search tree. Instead, they use queues which keeps the available tasks. The scheduling is performed by using the information associated with the queue, *e.g.*, the number of the elements of a queue. Schedulers based on this approach are called *queue-based* schedulers.

The scope of the research is restricted to tree-based schedulers and hereafter we refer to a tree-based scheduler just as a scheduler. For the design of an high performance scheduler, the following two issues must be clearly addressed:

1. how to efficiently identify and manage the load of each worker which changes dynamically almost at each creation of a node (choice point).

2. how efficiently for an idle worker to locate the available tasks.

As a matter of fact, the two issues conflict with each other. Different from conventional programs, logic programs require the larger amount of the scheduling activities. Moreover, the parallel load changes very frequently almost at every creation of a node. To identify and keep the accurate load is frequently prone to introduce high overhead. Without accurate information on the parallel load, to find the best node to explore becomes in turn a very complex and causes severe performance penalty. In this regard, the trade-off between accuracy and overhead which occurs in the identification of parallel load must be carefully assessed to achieve high performance scheduling. In the following discussion, we present a concise analytic model which provides a framework to analyze such overhead.

The analytic model is aimed at explaining the cost of scheduling within a tree based schedulers. In the parallel execution, a worker interacts with the others mostly in association of task switching. It is mainly because the position of a work in the search tree serves as valuable information in the scheduling and this information is in general maintained privately in each worker. Suppose that a worker w_s performs a scheduling and as a result, it finds an available work in the path being taken by w_d. The scheduling of w_s consists of a set of operations which can be explained in terms of their cost as follows:

1. A locating time (T_l) refers to the time spent in finding an available task from w_d.

2. A publishing time (T_p) refers to the time spent in making some private nodes public such that other workers can take some unexplored branches from the nodes.

3. An environment preparation time (T_e) refers to the time spent in making the environment of the new task for w_s. It is further classified as follows:

    - moving back time (T_b) refers to the time spent in making the environment of w_s be the same with that of the least common ancestor node between n_c and n_s.
    - moving forward time (T_f) refers to the time spent in updating the environment for the path between n_c and n_d in the memory of w_s.
    - synchronization time (T_s) refers to the time spent by either w_s or w_d in waiting until the other finishes the moving back or forward operation.

Depending on the parallel system, some of these activities are not necessary and the time spent in the activities are defined as zero. Under the model, the total overhead T for each instance of scheduling becomes

$$T = T_l + T_p + T_e = T_l + T_p + T_b + T_f + T_s.$$

Most tree-based schedulers have similar cost for T_l, T_g and T_p. However, the cost of the other activities is very different from each other depending on the memory model in the system. For example, T_e is very small in the Aurora because only the contents of the binding array are involved with the scheduling. In the Muse, as the entire stack is involved with the scheduling, the T_e is usually the larger than the one in the Aurora. In particular, the Aurora has T_s which is almost zero. This is because the moving operation (more precisely, deinstallation and installation of bindings in the binding) can be performed independent of other workers. On the other hand, the Muse must pay some cost when moving forward because w_d must wait during w_s copies the necessary portions of environments.

5.3.2 Common ancestor based approach

According the analytic model, our execution model eliminates the cost of the moving back operation by providing the Tagged Binding Array. As in the Aurora, the synchronization time is almost ignorable in our execution model. The publication time is almost the same in parallel logic systems because the operation is inherently independent of execution models. In these respects, only the location time becomes the objective of the optimization. Indeed, associated with runtime scheduling, the parallel support of our execution model is geared toward the minimization of the location cost.

Motivation and objectives

From the perspective of the tree based scheduling, one of important issues is how to efficiently identify relations which exist at runtime between workers. In the naive tree based scheduling framework, the information on the relative position is maintained in each node as a bitmap in which a bit is assigned to each worker. For example, when a worker is below the node, its bit in the bitmap contains '1'; otherwise, it contains '0'. This representation provides only the relative position between the workers, *i.e.*, below, same, and above. However, it does not provide such information as distance between workers. On the other hand, implementing some specific scheduling strategy is likely to incur high overhead because the scheduler have to examine some nodes. For example, a worker must examine all the live ancestor nodes and then visit a set of nodes for each worker to find any available task in the path being executed by the worker.

In the naive tree based scheduling, the overhead becomes severe because the size of the search tree is usually very large. When a worker tries to find some available work, it usually needs to walk around some nodes of some path. Also, when it needs to find the relative position in support of speculative scheduling heuristics, it also needs to walk around some nodes in search of the nodes below which the workers are executing. Without some global information, these procedures incur nontrivial overhead which may become larger as the search tree becomes larger.

In order to reduce the location cost, a more advanced scheduling framework is necessary which employs more efficient representation of the search tree other than the naive representation. This is particularly important for large-scale parallel logic programming systems since the total overhead to be paid for scheduling grows increasingly [8]. The underlying objective of the scheduling support in our parallel execution model is thus to explore a representation of the search tree which allows the scheduler to flexibly implement a variety of scheduling strategies with low overhead.

The least common ancestor relation between two workers

In order to achieve the objective, we use a new information, the *least common ancestor relation between workers*. When two workers w_1 and w_2 lie respectively on nodes n_1 and n_2, the common ancestor node between two workers w_1 and w_2 is defined as the least (youngest) common node n_c between nodes n_1 and n_2. Hereafter, we will call the least common ancestor node just the common ancestor for brevity. The main advantages of using the common ancestor node between two workers are as follows:

- The relative positions between workers are readily available.
- The distance between workers (or nodes in the workers) are directly available.

In order clarify the above argument, we provide an example. Consider three workers w_1, w_2, and w_3 which respectively stay on n_1, n_2, n_3. Let the common ancestor node between w_1 and w_2 be n_{12} and the common ancestor node between w_1 and w_3 be n_{13}. Provided that the OR-level of n_{12}, OL(n_{12}), is smaller than the OR-level of n_{13}, OL(n_{13}), w_2 is in the above of w_3. The distance between w_1 and w_2 becomes OL(n_1) - 2OL(n_{12}) + OL(n_2).

The common ancestor tree

With this advantages, the problem is how to identify the common ancestor nodes among the workers and how to represent them in the abstract machine level. One possible way will be to provide a table which is similar to the CANT (common ancestor node table) discussed in chapter 4. However, this is not absolutely suitable, because the common ancestor node table must be updated for each task switching and during the update, other workers must not access the table.

As an alternative, we represent the information in the form of a tree to be called a *common ancestor tree*. Figure 5.8 (a) shows an example common ancestor tree on which five workers are working. Under this representation, when a worker needs to extract information associated with other workers, it is sufficient to consider only a few nodes in the common ancestor tree. In this respect, the representation provides very high efficiency for general scheduling activities.

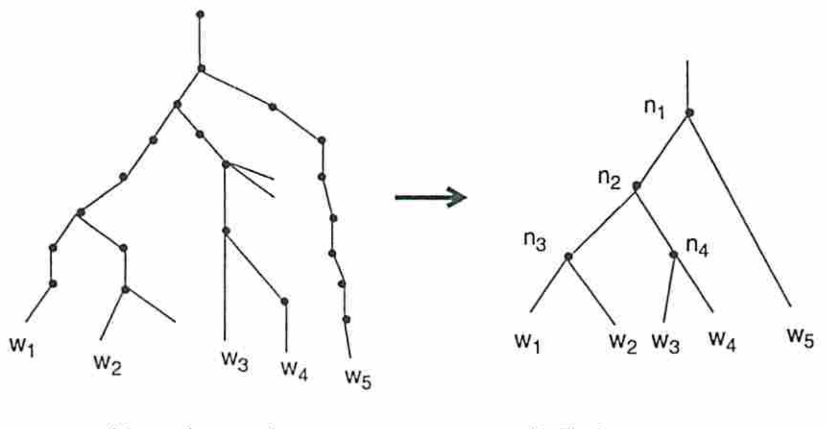


Figure 5.8: The naive search tree and its corresponding common ancestor tree

Parallel support for the common ancestor tree

The implementation of the common ancestor tree is done on top of the search tree. It is depicted in Figure 5.9. The only change in the ordinary search tree is that each node in the common ancestor tree has a pointer to the parent node and each worker has one register which keeps the pointer to the youngest node in the common ancestor tree.

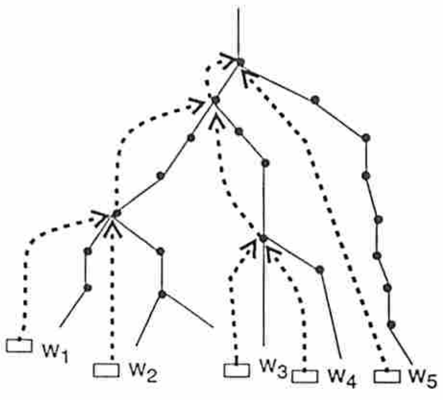


Figure 5.9: The implementation of the common ancestor tree

Figure 5.10 (a) provides a snapshot which depicts how the relative position between workers are implemented on top of the common ancestor tree. Associated with each common ancestor node, a bitmap is provided in which a bit is assigned to each worker and each bit indicates whether the corresponding worker is below or on the node.

Figure 5.10 (b) shows how the distance between two workers is calculated. It is very efficient since nodes in the common ancestor tree are inherently the common ancestor nodes between workers and the OR-node depths are readily available in our execution model.

Figure 5.11 shows how the leftmost checking is implemented on top of the common ancestor tree. Associated each worker, a branch stack is provided. Each entry of the stack has the branch number for the corresponding node. By comparing the values between two workers, the worker can identify which one is to left of it in the tree.

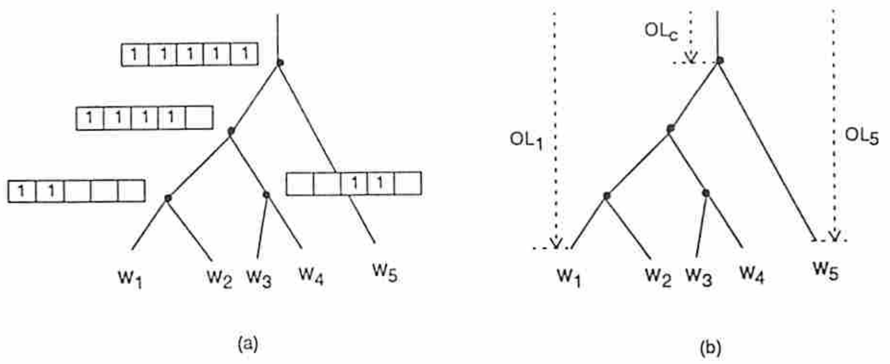


Figure 5.10: Representation of workers' relative positions under the common ancestor tree

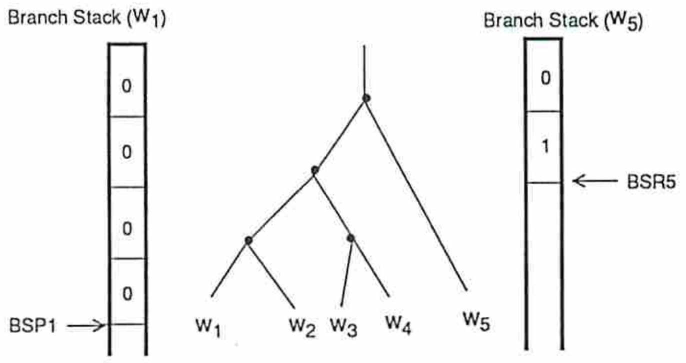


Figure 5.11: Left-right check under the common ancestor tree

5.4 Synthesis

This chapter presents our execution model. The objective underlying the design is to accomplish high flexibility and efficiency for the architectural and algorithmic optimizations in the runtime scheduling. The main components of the execution model are the Tagged Binding Array and a scheduling support based on the representation of the common ancestor tree. Compared with the Binding Array, the Tagged Binding Array is much efficient. Under the multiple binding environment based on our Tagged Binding Array, schedulers can accomplish low cost task scheduling because they do not have to carry out the deinstallation step in each task switching. On the other hand, the common ancestor tree provides a highly efficient representation of the runtime execution state of a program. It has an extremely simple form, maintaining only the essential part of the ordinary search tree. With the parallel support for the representation of the common ancestor tree, schedulers can flexibly implement a wide range of scheduling algorithms with minimal overhead. For reasons of these efficiency and flexibility, we believe that our parallel execution Model provides a good opportunity for high performance logic programming on large scale parallel machines.

Chapter 6

TCWAM: Translation-based Sequential Prolog Engine

This chapter presents a technique developed for the implementation of the sequential Prolog engine, named the TCWAM. The primary motivation of the TCWAM is to provide an experimental prototype which will be used as the sequential engine of our parallel logic programming system. In the TCWAM, we use a translated execution of Prolog programs rather than the emulation to obtain improved speed. This chapter presents the TCWAM and its performance.

6.1 Introduction

Despite of the efforts over the last decade, the implementation of Prolog is still an important issue in logic programming. Among a number of innovations, the WAM (Warren Abstract Machine) was one of breakthroughs which have contributed to the efficient implementation of logic languages. It has been a backbone in the implementation of many languages derived from the logic paradigm. Indeed, almost all the concurrent, constraint, and functional logic languages as well as Prolog, have been implemented by virtue of either a direct or an extended version of the WAM.

In the history of language implementation, the virtual machine approach is not new for logic languages. It is frequently used for the implementation of languages in several programming paradigms. The P-code for Pascal and the SECD machine for functional languages are the prominent examples. More recently, the implementation of Java language is also made via a well-defined virtual machine.

The process of compiling a program into code for a virtual machine and then converting it into the machine specific naive code usually provide an efficient means which facilities the language implementation on a target architecture. As the process of compilation is decoupled from the architectural specification, the compilation techniques and optimizations applied to the generation of the virtual machine code can be fully reused for all the architectures. Relieved of the concern about front-end compilation, language implementors can concentrate on the code generation.

The efficient naive code generation for modern RISC processors is still challenging. The performance of the generated code depends heavily on the optimizations applied to the code generation, while the code optimization has been a very hard and complex task. Most systems provide some high level languages compilers. Particularly, almost all the systems provide C compiler which can produce a very highly optimized code. Executing logic languages via C languages is thus an efficient way to achieves high performance with minimal effort. Along this line, the emulation of virtual machine instructions via C language has been most frequently used. With precise description of the virtual machine and the clean-cut instruction set, the implementation of an emulator in C is very simple.

However, software emulation of virtual machine instructions is absolutely slower compared with the the naive code. As an alternative, translating the virtual machine code into C code and then executing it after compiling by the C compiler have also been recognized as a promising way. By this, we can avoid the complexity of efficient code generation while obtaining very efficient code [29, 32]. It will be called a *translation based* approach and this chapter presents a new technique which addresses a variety of issues pertaining to the translation of PROLOG into C via the WAM.

The rest of the chapter is organized as follows. Section 6.2 provides an outline of the issues pertaining to the translation of PROLOG to C. Section 6.3 provides a review of the previous approaches. Section 6.4 presents the techniques developed for our TCWAM. Section 6.5 compares the TCWAM approach with others by means of some efficiency measures. Section 6.6 presents the performance of the TCWAM and its comparison with other implementations. Finally, section 6.7 concludes the chapter.

6.2 Issues in the Translation of Logic Programs

Code translation from a high level language into another language usually raises a number of issues which affect the performance of the generated code. Even though the language is compiled into well defined virtual machine code, provided the virtual machine instructions cannot be directly converted into a set of the high level language constructs, the procedure of the translation would become very complex. Besides, the produced code may become slower than the emulated version. This is very acute for the translation PROLOG to C via the WAM because the WAM code is flat with no procedural linkage. In this section, we thus identify the unit of instructions which needs to be addressed in the WAM code and the types of branches which transfer execution control, and briefly describe the issues associated with the translation.

We provide a patch of Prolog code along with its WAM code in Figure 6.1. Notice that in order to distinguish between the code for a predicate and the code for the first clause in the predicate, two different labels are used in the figure.

The sequence of instructions which need to be addressed in the WAM will be called an *addressable* and four kinds of addressables are defined without precise definitions as below:

- Predicate unit: A predicate call needs to address the starting location of the predicate code.
- Clause unit: Inside a predicate code, it is required to address the stating location of each clauses for recording the next alternative clause try (line 12).
- Indexing unit: The part of the predicate code which is responsible for indexing contains some branches either to the locations of clauses or some other locations inside itself. These are defined as indexing units.

```
a(X,Z) :- b(X,Y),c(Y,Z).  (1)  a: allocate       3       ; Predicate 'a'
 b(1,2).                  (2)       get_var      Y1,A1   ; Unification
 b(2,2).                  (3)       get_var      Y2,A2   ;  code
 c(2,3).                  (4)       put_val      Y1,A1   ; Argument
                          (5)       put_var      Y2,A2   ;  generation
                          (6)       call         b
                          (7)   x: put_val       Y2,A1
                          (8)       put_val      Y3,A2
                          (9)       deallocate
                          (10)      execute      c
                          (11) b:
                          (12) b0: retry_me_else  b1     ; Predicate 'b'
                          (13)      get_int       1,A1   ; clause b1
                          (14)      get_int       2,A2
                          (15)      proceed
                          (16) b1:       . . .           ; clause b2
                          (17)
                          (18) c:   get_int       2,A1   ; Predicate 'c'
                          (19)      get_int       3.A2
                          (20)      proceed
```

Figure 6.1: A patch of Prolog predicate and its WAM code

- Continuation unit: Given a predicate call, when the call succeeds, the control is transfered to the next instruction (for example line 7 after "call b" in line 6). The destination of the control is referred to as a continuation and the sequence of instructions not containing any branches or calls is denoted as the continuation unit. In Figure 6.1, line 7 - line 10 is an example.

In a program consisting of multiple modules[1], the scope of a branch will be sometimes made across modules. For example, a predicate call in a module, (*i.e.,* a reference to a predicate unit), is made to a predicate in another module. As to clause units, indexing units, and continuation units, the references are made always within a module.

In runtime, branches are carried out explicitly by a specific instructions such as "call" and "proceed" or implicitly by execution control such as "backtracking". As their destination locations, these branches use either the addresses offered as their operands or the contents of some registers. The former case corresponds to the direct addressing, while the latter corresponds to the indirect addressing. We will refer to the former as direct branches and the latter as indirect branches. The branches made to the above four addressables are classified according to the addressing type as follows:

- Branches to predicate units or indexing units are always direct branches.
- Branches to continuation unit are always indirect branches.
- Branches to clause units can be either direct or indirect.

With the existence of multiple modules, direct branches to predicates may occurs globally to other modules. The direct branches to indexing units or clause units always occur locally within a module. Indirect branches to continuation units or some clauses units are either local or remote. Therefore, four types of branches are classified as follows:

- Remote Direct Branch (RDB) to remote predicate units,
- Local Direct Branch (LDB) to indexing units or clause units,

[1]A module is defined as a part of program which is in a separate file and compiled separately from other modules.

- Remote Indirect Branch (RIB) to predicate, clause, or continuation unit, and
- Local Indirect Branch (LIB) to predicate, clause, or continuation units

The translation methods are mainly concerned with how to handle the above addressable units and branches. The main issues of translation are thus how to represent addressables in C and how to implement branches in C.

Representing an addressable and a branch respectively by a function and by a function call is a correct and will be the simplest way. However, it cannot be a satisfactory solution due to the following reason. The stack allocated at each entry of a function will grow exponentially because most C compilers do not perform tail-call optimization and call forwarding [27], which eventually limits the use of the system to only small toy programs.

6.3 Previous Approaches

A number of logic language systems are based on the translation approach. This section briefly reviews them with focus on the issues pertaining to execution control. In order to make the review clear and self-contained, we will provide the code resulting from the translation of the example Prolog program shown in the previous section. In the translated code, variable CP refers to the CP register in the WAM and variable ALT refers to the slot which contains the address of next alternative in a choice point.

6.3.1 KL1

KL1 is a stream AND-parallel logic programming language based on Flat GHC [75]. It has been implemented on a non-shared memory multiprocessor Multi-PSI/V2 [66].

As shown in the example code, each unit in a module is compiled uniformly into a function. Each function returns the address of the next function. Each module provides a *control* function which invokes a function returned by the previous function call. Now that a function, as a first class object, can be addressed in C language, at least all the addressables are correctly implemented. On the other hand, all the branches are uniformly performed via function calls.

```
module() {                          void pred_b() {
    while (PC) (*PC());                 ALT b1;
}                                       return CP;
                                    }
void pred_a(){
    push (CP)                       void b1() {
    CP=a01;                             return CP;
    return pred_b;                  }
}
                                    void pred_c  {
void a01() {                            return CP;
    pop (CP)                        }
    return pred_c;
}                                   void backtrack ()
                                        pop (ALT)
                                        return ALT
                                    }
```

Figure 6.2: Translated code in the KL1 system

This method is simple to implement. However, it is not satisfactory due to the following reasons. The overhead of C function calls caused from the allocation of stacks in the prologue of a function and deallocation in the epilogue is relatively large, compared with the very small code size of each function. In order to avoid the overhead, the later implementations of KL1 use a slightly different method which is similar to the one adopted by `jc` in the next subsection.

6.3.2 jc

`jc` [35] is a potable sequential implementation of Janus [71] which is designed for distributed constraint programming. As shown in the example code, each module is compiled into a single function in which each unit is defined as a code block[2].

[2]In this chapter, a part of a function in a C program is denoted as a code block.

```
main()  {

begin:
   switch(PC) {
         case a0:
     pred_a: push(CP);
              CP = a01; goto pred_b;
         case a01:
              pop(CP); goto pred_c;
         case b0:
     pred_b: PC = CP;
              ALT = b1, push(ALT); goto begin;
         case b1:
     clause_b1:
              PC = CP; goto begin;
         case c0:
     pred_c: PC = CP; goto begin;
   }
   backtrack:   pop(ALT); PC=ALT; goto begin;
}
```

Figure 6.3: Translated code in the jc system

Some local direct branches are implemented via goto statement which takes a label as its destination address. Indirect branches are implemented via switch statement. Both remote direct and indirect branches requires a specific interface between modules because goto statement of a function must take a label in the function. To provide the interface, each module has a table which keeps the address of each switch key. The tables are initialized at the start of the execution of a program. Because an address stored in the table is calculated dynamically with respect to its switch key, this method is referred to as "computed goto" [29]. In addition to the initialization cost, the remote branches becomes more expensive than the local branches. It should be noted that local branches are not also efficient because the cost of executing switch statement is more than 10 cycles in RISC machines.

6.3.3 Erlang

Erlang is a concurrent logic programming language designed for prototyping and implementing real-time systems [45]. The implementation translates each predicate unit into a function in which the other units appear as code blocks. Different from the previous approaches, Erlang system uses GNU C compiler, which benefits PROLOG to C translation with the following features. (1) The register clobbering makes it possible to declare some global variables as a set of machine registers. (2) Labels are treated as a first class object and thus it is possible to obtain the address of a label. (3) Goto statement is able to take a pointer, which allows an indirect branch to the content of a pointer. (4) Inlining an assembler code in C source programs is supported.

The translated code resulting from Erlang partially shows the potential GNU C languages as a vehicle for PROLOG translation. Each function is addressed through a reference to a label to support such units that are always locally addressed.

In order to avoid the allocation when a function called is made to a predicate unit, Erlang provides a global table which will contain the starting address of the code part for each predicate unit. To make such a table, each function is composed of an initialization part and a code part. The initialization part of a function is

```
a() {
                    table[a_id] = &&pred_a; return;

     pred_a:        push(CP);
                    CP = &&a01;  goto *table[b_id];
     a01:           pop(CP); goto *table[c_id];
}
b() {
                    table[b_id] = &&pred_b: return;

     pred_b:        ALT = clause_b1_id, push(ALT); goto *CP;
  clause_b1:        goto *CP;
}
c() {
                    table[c_id] = &&pred_c; return;

    pred_c:         goto *CP;
}

backtrack() {
                    table[backtrack_id] =  backtrack_id; return;
    backtrack_id:   pop(ALT); goto *ALT;
}
```

Figure 6.4: Translated code in the Erlang system

responsible for updating the global function table with the starting address of the code part. When the initialization part have been executed, the function will just return.

Although remote branches are efficient thanks to the global function table, one major drawback of the method is that the update of the global table must be carried out for all the exported predicates. Besides, because of the optimization of C compiler, a function may not be compiled in the code in which the initialization and the code part are physically separated. For example, due to some prevalent access of the global table in a function, the compiler may carry out an optimization such that some part of code is compiled into a common expression which will be executed both in the initialization and the code part. Consider that a compiler will place the common expression in front of the initialization part and pass the result in a register such that both the initialization part and the code part work on the value in the result. In this case, the register remains uninitialized when a code part is executed and causes incorrect execution. Some other problems associated with the global table are explained in some other paper [29].

6.3.4 WAMCC

The WAMCC [29] is one of the most complete and efficient translation based Prolog system. The code resulting from the translation is similar to the one resulting from KL1 in that all the addressables are compiled into separate functions. However, the branches in the WAMCC are not made to the entries of the functions. Assembly label is inlined just after the prologue with a function declaration in the header of the C source program. The branches are made to the position of the label to skip the prologue part.

The compiler then generates assembly code in which the label appears in the entry of a procedure call. Branches to a predicate unit are made through function calls whose function name is the label. It is implemented through assembly inlining of a bogus function name which is textually the same with the label. As a result, the control is always transferred to the entry point, skipping the prologue part.

Even though branches are implemented via a function call, it does not usually cause overhead because the function call in RISC machines are as efficient as jump

```
void pred_a();
void a01();
void pred_b();
void clause_b1();
void pred_c();
void l_backtrack();

void a() {
    asm("pred_a:");
    push(CP);
    CP=a01;
    pred_b();
}
void goal_a01() {
    asm(``a01:");
    pop(CP);
    pred_c();
}
```

```
 void b() {
        asm(``pred_b:");
       ALT = clause_b1;
        push (ALT); }
        (*CP)();
}
void b1() {
        asm(``clause_b1:");
        (*CP)();
 }
void c() {
        asm(``pred_c:";
        (*CP)();
 }
 void backtrack ()  {
        asm(``l_backtrack:'');
        pop(ALT);
        (*ALT)();
 }
```

Figure 6.5: Translated code in the WAMCC system

instructions. Now that all the addressables are compiled into separate functions, each module consists of a large number of functions. As a result, the object code contains relatively large amount of instructions which are compiled for prologues and epilogues for the functions. Besides, adopting to translate all the addressables by using direct branches, the WAMCC does not optimize some direct branch, which, implemented by a direct branch assembly instruction, will take less cycles than a function call.

6.4 Thread C-code WAM (TCWAM)

The TCWAM is a translation based Prolog system which we designed to use as the sequential engine of our parallel logic programming system. Its prototype was implemented on a PARISC machine and GNU C is used for the target language. In the design of the TCWAM we use the following three efficiency criteria

1. the speed of executable code
2. the size of produce object code
3. the time of compilation

The first criterion is the execution speed which should be a prime concern in the translation based execution. The second criterion is based on the following observation. Figure 6.4 depicts an assembly procedure for a C function with only one C statement, "printf("a")". In this code, it is noticed that seven instruction are used as for the prologue and epilogue of the function. As they are not executed at runtime, it is desirable to remove the prologue and the epilogue part from the produced assembly code. This can be accomplished by compiling the entire module into a single function. Finally, the time spent in compiling the produced C code is also considered. As noted in [29], the compilation time rapidly becomes larger as the size of a function becomes bigger. To reduce the compile time, it is essential to compile addressables into several functions rather than into a single function. A trade-off occurs between the object code size and the compilation time because criteria 2 and 3 cannot be satisfied at the same time.

```
.PROC
.CALLINFO FRAME=128,CALLS,SAVE_RP,SAVE_SP,ENTRY_GR=3 ; Prologue
stw %r2,-20(0,%r30)
copy %r3,%r1
copy %r30,%r3

(code for printf(a))

ldw -20(0,%r3),%r2
ldo 64(%r3),%r30
ldwm -64(0,%r30),%r3
bv,n 0(%r2)

.ENTRY
.EXIT
.PROCEND
```

Figure 6.6: Prologue and epilogue code generated by gcc 2.6.3 on a HP's SPP IAX-0016

In the presence of the trade-off between object code size and compilation time, we opted for the smaller code size due to the following reasons. In a system which translates an entire module into a single function, the size of a function can be controlled by making a Prolog program in multiple modules. Indeed, according to our experiment, the compilation time does not show much difference for a C function whose size is in the order of hundred lines of a C function.

In the TCWAM, in order to reduce the compile time and the required space, the produced object code for each addressable unit is compiled into an assembly procedure. To this end, two macro are defined which correspond to the assembly directives for the entry and the exit of an assembly procedure. Besides, two more macros are provided to define labels. In C programs, each occurrence of such a label makes the previous code as an assembly procedure and provides a procedure head to make the subsequent code become a new assembly procedure.

Figure 6.4 shows the contents of directives and the C program resulting from a translation. In this code, the remote branches are implemented via function calls as in the WAMCC. The global addressables are implemented with the inlining assembler labels. The local branches, many of which appear in relation with indexing, are translated into "goto" statements. As noted earlier, because they are translated either to direct "branch" or "jump" instructions, the "goto" statement is slightly more efficient than function calls.

6.5 Qualitative Comparison

In this section, we compare the translation methods with focus on implementation techniques and their performance impact.

Table 6.5 shows a summary of how a module, a predicate, and a clause are mapped into C functions and a summary of how branches are implemented.

Table 6.5 shows the result obtained from qualitative comparison of the systems made in terms of the three efficiency criteria. The efficiency of code is defined as "low" if a module is compiled into a single function because each function in the produced code contains the Prologue and epilogue code; otherwise, it is defined as "high". The compilation time is defined as "low" if a module is compiled into

```
#define M_Proc_Head
    asm(".PROC");
    asm(".CALLINFO  FRAME=128,CALLS,SAVE_RP,SAVE_SP,ENTRY_GR=4");
    asm(".ENTRY");
#define M_Proc_Tail
    asm(".EXIT");
    asm(".PROCEND");
#define M_Global_Label(name)
    __name:
    asm(".EXPORT " _name ",ENTRY,PRIV_LEV=3,RTNVAL=GR");
    M_Proc_Tail
    asm("\x0a" _name);
    M_Proc_Head
    asm(" "::"g"(M_C_ASymbol(name,c)));
#define M_Local_Label(name)
    __name:
    M_Proc_Tail
    asm("\x0a" _name);
    M_Proc_Head
    asm(" "::"g"(__name,c)));

void pred_a();                void pred_b()
                              void pred_c();
void module1()                void module_2()
{                             {
    asm("pred_a:");               M_Global_Label(pred_b)
    push(CP);                     ALT = &&clause_b1;
    CP=&&a01;                     push (ALT); }
    pred_b();                     (*CP)();
    M_Local_Label (a01)           M_Local_Label(b1) {
    pop(CP);                      (*CP)();
    pred_c();
}                                 M_Global_Label(pred_c)
                                  (*CP)();
                                  Global_Label(backtrack)
                                  pop(ALT);
                                  (*ALT)();
                              }
```

Figure 6.7: Assembly code generated by gcc 2.6.3 on a HP's SPP IAX-0016

System	Module	Predicate	Clause
Janus	One function	One code block	One code block
KL1	Multiple functions	Multiple functions	Multiple functions
Erlang	Multiple functions	One function	One code block
WAMCC	Multiple functions	Multiple functions	Multiple functions
TCWAM	One function (multiple assembly procedure)	One code block (multiple assembly procedure)	One code block (multiple assembly procedure)

Table 6.1: The produced C-code for addressables. As noted earlier, a code block refers to part of code inside a function. In case of the TCWAM, the structure of the produced assembly code is explained as well.

System	RDB	GIB/LIB	LDB
Janus	Jump to label	Swith to computed label	Switch to computed label
KL1	Function call	Function call	Functional call
Erlang	Indirect jump to recorded label	Indirect jump to recorded label	Indirect jump to recorded label
WAMCC	Function call (inlined label)	Functional call (inlined label)	Function call (inlined label)
TCWAM	Function call (inlined label)	function call (inlined label)	Function call label

Table 6.2: Control mechanism: A computed label refers to a constant used as a key for selecting the label, a recoded label refers to the content of the global table, and an inlined label refers to a label created through the assembly inlining.

System	Code size efficiency	Compilation time	Inter-modular efficiency
Janus	Low	Low	Low
KL1	Low	High	High
Erlang	High	Low	Low
WAMCC	Low	High	High
TCWAM	High	Low	High

Table 6.3: Comparison of translation methods

a function; otherwise, it is defined as "high". If the remote branches are more expensive than the local branches, the inter-module efficiency is defined as "low"; otherwise, it is defined as "high".

According to Table 6.5, no one translation method satisfies all the three efficiency criteria; they sacrificed at lease one criterion.

However, it should be noted that the three criteria do not directly determine the execution speed. In terms of execution speed, which is the most important criteria, the technique employed in the WAMCC and the TCWAM result in the highest speed [29].

6.6 Performance Evaluation

The previous sections presents issues pertaining to the translation and the techniques developed in the implementation of logic languages. This section presents the performance of the translation technique developed and implemented in the TCWAM.

In order to frame the evaluation of the performance, we ported the WAMCC on a HP's SPP IAX system. Originally, the TCWAM is developed on a SPP IAX system as the sequential Prolog engine for use in our parallel logic programming prototype. Because both the WAMCC and the TCWAM are Prolog compilers based on the WAM, we can make a reasonable comparison to identify the advantageous and disadvantageous of the two translation methods.

Prolog Program	Lines	Assembly code size	Object code size	Executable code size	Comp. time	Exe. time
boyer	395	283	63	266	55	1374
browse	111	79	20	237	12	1662
cal	202	75	19	237	11	180
chat_parser	1184	794	182	356	619	333
crypt	96	59	15	237	8	13
ham	90	61	16	233	8	1875
meta_qsort	146	71	18	238	9	21
nand	574	431	95	299	191	60
nrev	105	41	11	233	5	277
poly_10	112	71	18	238	8	109
queen(16)	95	28	8	229	4	1321
queens_8	79	41	11	233	6	331
queens_10	79	41	11	233	6	6018
reducer	388	217	51	262	37	100
sdda	327	141	34	250	21	6
sendmore	66	59	14	233	8	139
tak	35	20	6	229	4	298
tak_gvar	54	26	8	229	5	10
zebra	57	42	12	233	7	112

Table 6.4: Evaluation results of the TCWAM: assembly, object and executable code are in Kbytes, compiler time is in seconds, and execution time is in milliseconds.

Prolog Program	Assembly code size	Object code size	Executable code size	Compilation time	Execution time
boyer	1.48	1.72	1.43	1.04	1.41
browse	1.40	1.48	1.37	1.25	1.38
cal	1.75	1.91	1.38	1.45	1.23
chat_parser	1.63	1.97	1.61	0.54	1.20
crypt	1.78	1.95	1.37	1.62	1.00
ham	1.52	1.59	1.37	1.50	1.23
meta_qsort	1.44	1.58	2.05	2.11	1.48
nand	1.57	1.95	1.48	0.82	1.37
nrev	1.44	1.51	1.35	2.00	1.32
poly_10	1.56	1.69	1.36	1.75	1.50
queens(16)	1.43	1.34	1.36	1.50	1.29
queens_8	1.34	1.34	1.35	1.50	1.21
queens_10	1.34	1.34	1.35	1.50	1.21
reducer	1.63	1.91	1.44	1.43	1.26
sdda	1.62	1.89	1.39	0.62	1.67
sendmore	1.68	1.81	1.39	0.62	1.37
tak	1.30	1.15	1.34	1.75	1.15
tak_gvar	1.34	1.23	1.34	1.20	1.50
zebra	1.43	1.39	1.35	1.43	1.11
average	1.51	1.62	1.43	1.35	1.31

Table 6.5: Comparison: TCWAM versus WAMCC (WAMCC/TCWAM)

Table 6.4 shows the evaluation results for the TCWAM. The assembly code size is obtained with the compiler option "-S" in gcc version 2.6.3. The object code is the output of the "gcc" compiler obtained with the compiler option "-c".

Table 6.6 shows the results of comparison between the TCWAM and the WAMCC. For almost all the evaluated items, the TCWAM shows improvement over the WAMCC. In the assembly code size, object code size, and the executable code size, the TCWAM shows improvements respectively by 51, 62, and 43 percents. Also, the average improvement of the execution speed is 31 percents. The TCWAM shows the improvement of compilation time by 35 percents. However, a close examination of the values shows that for all the Prolog programs with bigger

Prolog Program	Assembly code size	Object code size	Executable code size	Execution time
boyer	1.23	1.14	1.14	1.29
browse	1.32	1.15	1.14	1.47
cal	1.13	1.05	1.12	1.15
chat_parser	1.27	1.19	1.16	1.40
crypt	1.37	1.13	1.12	1.00
ham	1.20	1.00	1.12	1.41
meta_qsort	1.24	1.06	1.12	1.43
nand	1.37	1.26	1.16	1.35
nrev	1.29	1.09	1.12	1.45
poly_10	1.29	1.11	1.12	1.39
queens(16)	1.28	1.01	1.13	1.25
queens_8	1.29	1.09	1.12	1.45
queens_10	1.29	1.09	1.12	1.48
reducer	1.32	1.19	1.14	1.41
sdda	1.35	1.23	1.13	2.17
sendmore	1.32	1.14	1.43	1.30
tak	1.25	1.01	1.16	1.27
tak_gvar	1.27	0.90	1.42	1.30
zebra	1.26	1.08	1.39	1.32
average	1.28	1.10	1.18	1.38

Table 6.6: Comparison: TCWAM versus TCWAM-NCR

sizes, the compilation of the TCWAM shows roughly two times slower than the WAMCC.

Table 6.6 shows the result of comparison between the version with clobbered registers (TCWAM) and a version without clobbered registers (TCWAM-NCR). In the version with clobbered registers, total 10 registers, (r10, r11, r12, r13, r14, r15, r16, r17, r18,, r19), are mapped to the WAM registers. For all the evaluation items, the TCWAM shows improvement over the TCWAM-NCR. With regard to the assembly code size, object code size, and the executable code size, the TCWAM shows improvements respectively by 28, 10, and 18 percents. Besides, the average improvement of the execution speed is 38 percents.

Prolog Program	TCWAM 1.00	WAMCC 2.21	BinProlog 3.0	XSB-Prolog 1.4.0	SWI-Prolog 1.8.11
boyer	1.00	1.41	2.74	4.68	8.66
browse	1.00	1.38	2.72	4.07	6.24
cal	1.00	1.23	3.78	5.84	21.05
chat_parser	1.00	1.20	1.47	2.19	2.51
crypt	1.00	1.00	1.06	2.50	6.25
ham	1.00	1.23	1.50	2.51	3.59
meta_qsort	1.00	1.48	3.28	4.59	4.26
nand	1.00	1.37	3.64	0.00	4.78
nrev	1.00	1.32	1.15	2.29	7.38
poly_10	1.00	1.50	2.09	3.59	5.98
queens(16)	1.00	1.29	2.46	3.41	16.45
queens_8	1.00	1.21	1.60	2.71	5.99
queens_10	1.00	1.21	1.42	2.53	4.98
reducer	1.00	1.26	2.57	0.00	4.34
sdda	1.00	1.67	3.33	5.56	3.33
sendmore	1.00	1.37	6.54	3.98	15.33
tak	1.00	1.15	2.92	2.98	13.58
zebra	1.00	1.11	1.70	2.26	2.47
average	1.00	1.30	2.55	3.48	7.62

Table 6.7: Improvement rate of the execution speed for the generated code (a)

Prolog Program	TCWAM 1.00	Sictus 2.1 emulated	Sictus 2.1 native	Quintus 2.5.1	Aquarius
boyer	1.0	2.43	1.04*	1.16	0.89*
browse	1.0	2.28	1.44*	1.15	2.11*
cal	1.0	3.66	0.45*	2.06	0.84*
chat_parser	1.0	1.38	1.64*	0.79	2.34*
crypt	1.0	1.69	1.23*	1.06	1.60*
ham	1.0	1.43	1.69*	0.85	3.71*
meta_qsort	1.0	1.57	1.45*	1.64	2.03*
nand	1.0	2.28	1.05*	1.48	2.20*
nrev	1.0	1.39	2.39*	0.55	2.84*
poly_10	1.0	1.60	1.34*	1.25	2.87*
queens(16)	1.0	2.12	1.48*	1.49	3.11*
queens_8	1.0	1.70	1.56*	1.01	4.43*
queens_10	1.0	1.61	1.56*	0.95	5.02*
reducer	1.0	1.26	1.79*	1.26	2.14*
sdda	1.0	2.56	0.56*	1.89	0.90*
sendmore	1.0	3.98	0.99*	1.66	2.10*
tak	1.0	2.13	1.23*	3.38	7.99*
zebra	1.0	1.28	1.02*	0.98	1.47*
average	1.0	2.02	1.33*	1.37	2.70*

Table 6.8: Improvement rate of the execution speed for the generated code (b): the values of Sictus 2.1 and Aquarius are the improvement over the TCWAM, which are indicated by stars.

6.7 Synthesis

This chapter presents a technique developed for the implementation of the sequential Prolog engine which we call TCWAM. With regard to the translated code size and the execution time of the code, the TCWAM shows about 30 percents improvement over the WAMCC, which is up to now the most efficient translation based sequential Prolog system.

The comparison of the execution speeds between the TCWAM and some academic and commercial Prolog system indicates that the TCWAM is faster than the most sequential Prolog system except Sictus 2.1 (naive) and Aquarius. Considering that Scitus 2.1 (naive) generates the naive code and Aquarius is the fastest sequential Prolog system, the translation technique which we developed and implemented for the TCWAM is apparently quite successful.

Chapter 7

Flat indexing: an Indexing Technique for OR-parallel Logic Programming

Indexing is a method which prunes away unnecessary inferences in the evaluation of logic programs. Usually, it is implemented by a set of abstract instructions. To find an optimal indexing is quite complex and also demands a large number of abstract instructions for the implementation. In the indexing scheme of the WAM, the first argument of a clause is used as a key for indexing. Viewed from the standpoint of a trade-off between efficiency and simplicity, using the first argument as a key is quite a reasonable choice. However, in the indexing scheme, an invocation of a predicate sometimes results in the creation of two contiguous choice points in a search path. In this case, the OR-parallelism is expressed in the two choice points. Therefore, viewed from the OR-parallel execution standpoint, the indexing scheme of the WAM is not efficient in terms of parallelism exposition. In order to enhance the parallelism exposition, we developed a new efficient indexing scheme which we call *flat indexing*. This chapter presents the flat indexing scheme.

7.1 Introduction

Logic languages based on the SLD refutation [61] impose a strictly sequential search over the list of clauses which make up a predicate. In the search, all the clauses in the list are *tried* for a given goal. In each *clause try*, unification occurs between the clause's arguments and the goal's arguments. If all the arguments of the goal are unbound variables, all the clauses must be tried because every clause will be successfully unified. When some of the arguments are partially instantiated, the information can be used to prune away some clauses from the list of clauses to try, because it allows us to determine that some clauses always produce unification failure. The technique is usually called *indexing* and the indexing is clearly a good way to improve the performance when the number of clauses making up a predicate is large. Therefore, most logic language compilers produce a code which supports indexing [29].

The WAM has been dominantly used as the sequential engine in parallel implementations of logic languages [5, 15]. By doing that, parallel systems can benefit from the optimizations developed for the WAM and thus preserve high single thread performance. In such systems, the virtual machine structure and instructions provided in the WAM are slightly modified or extended in association with parallelism. The indexing scheme of the WAM and its associated instruction set are also slightly modified.

In the execution of a WAM code, a choice point is created in memory as a unit of information used for the management of the execution of a predicate [78]. It keeps the next alternative and some other information such as goal arguments. In parallel execution, all the OR-parallelism available for a predicate is thus exposed in a choice point. However, close observation shows that two choice points are sometimes created for a predicate in a search path. This phenomenon is rooted from the indexing scheme. In the sequential execution of a WAM code, it is a very trivial issue because the indexing scheme produces very compact code, which is in fact one of the design objectives of the WAM.

However, within the context of parallel execution, the indexing scheme in the WAM is inefficient in terms of parallelism and scheduling. In this chapter, we investigate the problem in detail and suggest a new efficient indexing scheme which we will call *flat indexing*. Moreover, in order to verify the performance, we have implemented both indexing schemes in a sequential Prolog system and evaluated the number of choice points created for a set of benchmarks. This chapter will thus presented the evaluation result both for the indexing scheme of the WAM (which will be called the *WAM indexing*) and for the flat indexing. The rest of this chapter is organized as follows: Section 7.2 gives a brief introduction of the logic languages and the principles of the WAM indexing scheme as a background. Section 7.3 presents the analysis framework for indexing schemes and the analysis result of the WAM indexing scheme in terms of parallelism. Section 7.4 presents the flat indexing scheme. Section 7.5 reports the evaluation results. Finally, section 7.6 concludes the chapter.

7.2 Review of the WAM indexing

To make this chapter self-contained we introduce the principles of the WAM indexing scheme along with some terminology and definitions for indexing.

In spite of the research efforts expended over the last decade, the implementation of Prolog is still an important issue in logic programming. Among a number of innovations, the WAM (Warren Abstract Machine) was one of breakthroughs which have contributed to the efficient implementation of logic languages.

Indexing is a method which prunes away unnecessary inferences in the evaluation of logic programs. Usually, it is implemented by a set of abstract instructions. To find an optimal indexing is quite complex and also demands a large number of abstract instructions for implementing it. In the WAM indexing scheme, the *first argument of a clause* is used as the *key*. According to the usual programmers' tendency, clauses making up a predicate are usually defined differently depending on data types. The differentiation is mostly reflected in the first argument. Considering the trade-off between efficiency and simplicity, the usage of the *first argument* as the *key* for indexing is a quite reasonable choice.

The WAM supports four data types: *variable, constant, list*, and *structure.* When the key is a variable, the key will always unify with an input argument of any type. The WAM indexing scheme is applied to each predicate. For the set of clauses, $(c_1, \ldots, c_n)$, which make up a predicate, it starts with grouping them into a set of contiguous *partitions*, $P_1, \ldots, P_m$ $(1 \leq m \leq n)$, where each subsequent P_i is

- (type α) either a single clause whose key is a variable or

- (type β) a maximal subsequence of contiguous clauses whose key is not a variable.

```
c1:      match( sum(A,B), sum(C,D))  :- match(sum(A+D-1), sum(C+B-1)) .
c2:      match( sum(A,B), C )        :- match(B-1, sum(C,B-1)).
c3:      match( a, b)                :- match(numeric(a), numeric(b)).
c4:      match( X, ascii(Y) )        :- match(ascii(X), digit(Y)).
c5:      match( a, b )               :- match(ascii(a), ascii(b)).
c6:      match( a, b )               :- match(digit(a), digit(b)).
c7:      match( b, X )               :- match(digit(b), digit(X)).
c8:      match( sum(A,B), sum(C,D) ) :- equal((A-C), equal(D-B)).
c9:      match( sum(A,B), C )        :- match(sub(C-A), B).
c10:     match( [A,B], [C,D] )       :- match(A, C), match(B,D).
c11:     match( [a,A], [C,b] )       :- match(a,C), match(A,b).
c12:     match( X, numeric(Y) )      :- match(numeric(X), numeric(Y)).
```

Figure 7.1: Clauses making up predicate "match/2"

Figure 7.1 presents an example Prolog program. It shows twelve clauses which define predicate "match/2". According to the partitioning rule discussed above, the clauses are grouped into four partitions, $P_1 = \{c_1, c_2, c_3\}$, $P_2 = \{c_4\}$, and $P_3 = \{c_5, c_6, c_7, c_8, c_9, c_{10}, c_{11}\}$, $P_4 = \{c_{12}\}$.

Partitioning is introduced just for making a compact code by excluding the clauses, whose key is a variable, from indexing. Partitioning is just a preliminary step for indexing; as a matter of fact, they are always visited regardless of the data type of the input key. In the WAM code, these partitions are chained such that each subsequence is visited consecutively whenever the predicate is called. The code which implements the chaining of the partitions is based on "try_me_else", "retry_me_else, and "trust_me_else_fail" [78]. Figure 7.2 shows a skeleton of the code which contains the chaining of those partitions defined for `match/2`.

The actual indexing is implemented for each partition. A partition of type α does not have any indexing because it has only one clause which must always be tried because its key is a variable. A partition of type β consists of one or several clauses. The key of each clause can be only one of the following three data types:

```
match/2     try_me_else P2_label
            [indexed code for P1]
P2_label    retry_me_else P3_label
            [indexed code for P2]
P3_label    retry_me_else P4_label
            [indexed code for P3]
P4_label    trust_me_else_fail
            [indexed code for P4]
```

Figure 7.2: Patitions and its structure in the generated code

constant, list, and *structure*. They are grouped into four *subpartitions*, S_c, S_l, S_s, and S_t, where S_c (*resp.* S_l, S_s) is an ordered set of clauses whose key is a constant (*resp.* a list, a structure) and S_t is the set of all the clauses which belong to the partition.

For each partition of type β, the WAM indexing produces code which will work as follows. For a given input argument, according to its data type, one of the subpartitions is selected. If the input argument is a constant (*resp.* list, structure, variable), then S_c (*resp.* S_l, S_s, S_t) is selected. When either S_l or S_t is selected, all the clauses in the subpartition will be tried because both a list and a variable do not have any specific value. In case of either subpartition S_c or S_s, one more level of indexing is still possible, because multiple different values can be defined as a key for clauses in subpartitions S_c or S_s. In this case, only those clauses in the subpartition whose key value matches with the value of the input argument are selected. The subpartition or a subset selected in the above procedure will be referred to as a *bucket*. S_l and S_t are always defined respectively as a bucket since no more indexing is made for them; however, a subset of S_c or S_s can be defined as a bucket according to the above discussion.

The selection and dispatching of a subpartition is implemented by instruction *switch_on_term*. The four arguments of the instruction specifies the destination of dispatching for the input argument respectively of *variable*, *constant*, *list*, and *structure* data type, as explained below.

- **Variable:** the designation becomes the first clause and eventually all the clauses in S_t will be tried.
- **List:** the destination becomes
  - the clause whose key is a *list* if only one such clause exists, or
  - the starting location of the bucket for *list* data type if more than one clauses are defined whose key is of *list* data type, or
  - the starting location of the failure code if no clause exists with its key of *list* data type.
- **Constant** (*resp.* **Structure**): the destination becomes
  - the clause whose key is a *constant* (*resp. structure*) data type if only one such clause exists, or
  - the location of *switch_on_constant* (*resp. switch_on_structure*) if more than one clauses are defined whose key is of *constant* (*resp. structure*) data type, or
  - the starting location of the failure code if no clause exists with a key of *constant* (*resp. structure*) data type.

S_t is implemented in the same way which was used for implementing partitions. For S_c or S_s, more than one buckets may exist. Instruction *switch_on_constant* or *switch_on_structure* selects a bucket. In these instructions, the selection of a bucket is made by using a hash table in which a hash entry is provided uniquely for each value and has the address of the corresponding bucket. If a bucket consists of more than one clauses, it is implemented by instructions *try*, *retry*, and *trust*. Figure 7.3 depicts the structure of the code for an emample partition with k clauses. Figure 7.4 shows the corresponding code for partition P_3 of predicate *match/2* shown in Figure 7.1.

Before we proceed to the next section, let us briefly describe the relation between the WAM indexing scheme and the choice points created for each invocation of a predicate. The principal role of a choice point is to keep the information required for the execution of a predicate. It is thus rather natural to provide one choice point per each invocation of a predicate. However, sometimes two choice points are allocated under the WAM indexing.

```
Pi code:
              switch_on_term Ci1, Cia | C_Switch | fail, Cib
                                | L_Bucket | fail, Cic | S_Switch | fail

C_Switch:     switch_on_constant [pointers to buckets]
                Lists of buckets for constants

L_Bucket:     A bucket for lists

S_Switch      switch_on_structure [pointers to buckets]
                Lists of buckets for structures

Ci1:          Code for clause Ci1
Ci2:          Code for clause Ci2
   ...
Cik:          Code for clause Cik
```

Figure 7.3: A structure of the code for a partition which has k clauses

```
P2 code:      switch_on_term C5_Label, C_Swich, L_Bucket, S_Switch

C_Switch:     switch_on_constant 2, {a: C_a_Bucket, b: C7_Code)
C_a_Bucket:   try   C5_Code
              trust C6_Code
L_Bucket:     try   C10_Code
              trust C11_Code
S_Switch:     switch_on_structure 1, {sum/2 : S_sum_Bucket}
S_sum_Bucket: try   C8_Code
              trust C9_Code

C5_Label:     try_me_else C6_Label
C5_Code:      Code for 'match( a, b ) :- match(ascii(a), ascii(b)).'

C6_Label:     retry_me_else C7_Label
C6_Code:      Code for 'match( a, b ) :- match(digit(a), digit(b)).'

C7_Label:     retry_me_else C8_Label
C7_Code:      Code for 'match( b, X ) :- match(digit(b), digit(X)).'

C8_Label:     retry_me_else C9_Label
C8_Code:      Code for 'match( sum(A,B), sum(C,D) ) :- equal((A-C),
                                                        equal(D-B)).'

C9_Label:     retry_me_else C10_Label
C9_Code:      Code for 'match( sum(A,B), C ) :- match(sub(C-A), B).'

C10_Label:    retry_me_else C11_Label
C10_Code:     Code for 'match( [A,B], [C,D] ) :- match(A,C),
                                                 match(B,D).'

C11_Label:    trust_me_else_fail
C11_Code:     Code for 'match( [a,A], [C,b] ) :- match(a,C),
                                                match(A,b).'
```

Figure 7.4: A code for partition P_3 of "match/2"

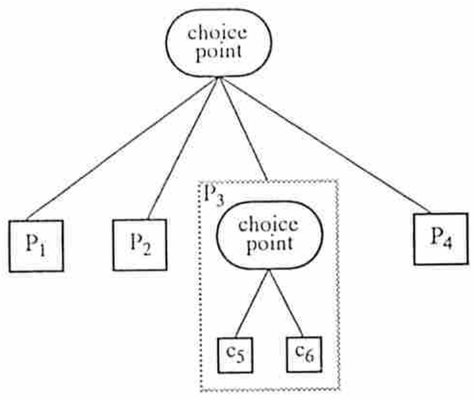


Figure 7.5: Choice points and parallelism

In the WAM, instructions *try_me_else* and *try* create respectively a choice point. When more than one clause is defined for a predicate, the code for the first clause always starts with instruction *try_me_else*. In this case, if a bucket with more than one clause exists in any of the partitions, its code starts with *try* instruction. Therefore, if the bucket is selected at runtime, two choice points are created in memory respectively by *try_me_else* and *try* instructions.

For example, in the execution of *match/2*, instruction *try_me_else* creates a choice point for which four partitions are exposed as alternatives branches (Figure 7.5). In the sequential execution, the four partitions are executed sequentially from left to right. In parallel execution, they can be executed in parallel respectively by other processors. The figure also depicts the case that a new choice point is created in partition P_3 by *try* instruction in `C_a_Bucket` (Figure 7.4), when the input argument, which matches with the key, is "a".

7.3 Analysis of the WAM Indexing Scheme

In this section, we present an analysis which aims at identifying the influence that the WAM indexing has on the OR-parallelism of a Prolog program. The analysis consists of the identification of the shape of a search tree created under the WAM indexing scheme and also a quantitative evaluation of the amount of OR-parallelism.

In the sequential execution of a WAM code, the creation of two contiguous choice points for a predicate can be regarded just as a variance of an implementation since it does not cause any notable performance penalty. In parallel execution, however, it has very harmful influence on the performance by affecting the amount of parallelism per choice point and the efficiency of task scheduling.

For a predicate P defined by more than one clauses, we now provide some notations and definitions associated with the WAM indexing. Let the set of clauses which make up the predicate be $c_1, \ldots, c_n$. Suppose that the indexing scheme produces m partitions, $P_1, P_2, \ldots, P_m$. Let us define a mapping N such that $N(S)$ be the number of elements of a set S. The number of clauses in partition P_i is then denoted as $N(P_i)$, *i.e.*, $\sum_{i=1}^{m} N(P_i) = n$. For a partition P_i, let the buckets to be defined for an input argument be b_{v_i}, b_{l_i}, b_{c_i}, and b_{s_i}, respectively for a variable, a list, a constant, and a structure. According to the definition of a bucket, a bucket created for either *constant* or *structure* data is for a specific value and it does not normally contain all the clauses whose key is of the type. The partition and its buckets thus have the following relation.

$$N(P_i) \geq N(b_{l_i}) + N(b_{c_i}) + N(b_{s_i})$$

In this chapter, we will represent a predicate by a tree called an *indexing tree*. The indexing tree for the execution of a predicate is informally defined as follows. Choice points resulting from the execution of a predicate are defined as nodes in the tree. These nodes will be called *cp-nodes*. In the execution of a predicate, if m, the number of partitions, is larger than one, a choice point is always created at the beginning of the execution. In this case, the cp-node becomes the root of the predicate's indexing tree and it will have m edges. To each edge, an indexing tree defined for the corresponding partition is connected as a subtree. For each partition, one of the following three case may occur as the result of *switch_on_term*, *switch_on_constant*, or *switch_on_structure*:

1. No one clause is tried.

2. One clause is tried.

3. A bucket is chosen which has more than one clauses.

For each case, a new node is connected to the edge. The node for case 1 or 2 is called a terminal node (*t-node*) and does not have any subtree below it. On the other hand, a choice point will be always created in case 3 because the number of clauses in the chosen bucket b_i is more than one. As a result, a subtree is attached to the edge which consists of a new cp-node with $N(b_i)$ t-nodes as its child nodes. Shown from the above discussion, the indexing tree of a predicate can have two levels, each respectively for partitions and for buckets in each partition. Table 7.1 illustrates the form of an indexing tree for a predicate.

Given a predicate, the indexing tree will be in a different form depending on the data type of an input argument because the form of an indexing tree of a predicate depends on which clauses will be executed at runtime. If the input argument is a variable or a list, the number of cp-nodes for the indexing tree will be always the same. On the other hand, if the input argument is a constant or a structure, the indexing tree has a different form depending on the value of the input argument. In this case, we can derive the maximum case and also minimum case of the indexing tree.

- The maximum case occurs when the number of cp-nodes is the maximum. It is defined as follows. Given an input argument, let B_i be the bucket chosen in the partition P_i. Also, let the number of bucket b_i for all i $(1 < i < m)$, which has more than one clause, be r. The number r is always uniquely defined for an input value. When the input is the constant (*resp.* the structure) which produces the maximum number of r, let us denote its corresponding bucket in a partition P_i $(1 < i < m)$ as b'_{c_i} (*resp.* b'_{s_i}). In this case, the number of cp-nodes in the resulting indexing tree will be defined as the maximum number of cp-nodes.
- The minimum case occurs when the number of cp-nodes is the minimum. It is defined as the indexing tree created when the input value, whose type is either a constant or a structure, does not match with any of the key values.

Table 7.1 summarizes both the maximum and minimum cases for each data type. In the table, p stands for the number of choice points created with respect to partitions. Therefore, it becomes "0" if the number of partition is "1"; otherwise, it becomes "1". Defined respectively for the data type of the input argument, r is the maximum number of buckets which have more than one clauses. Now that a choice point is created for each such bucket, the total number of cp-nodes becomes p plus r. The number t-nodes is calculated as follows: The total number of partitions minus the number of cp-nodes (*i.e.*, m-r) becomes the number of t-nodes in the first level, and the summation of clauses in all the buckets of each partition (*e.g.*, $\sum_{i=1}^{m} N(b'_{c_i})$ for *constant* data corresponds to the number of t-nodes in the second level. The summation of the t-nodes of the two levels thus becomes the total number of t-nodes. As discussed earlier, the table shows that the maximum and minimum cases are the same when the input argument is either a variable or a list.

	Maximum case		Minimum case	
Input Argument	Number of cp-node	Number of t-nodes	Number of cp-node	Number of t-nodes
Variable	p	n	p	n
Constant	$p+r$	$m\text{-}r+ \sum_{i=1}^{m} N(b'_{c_i})$	p	m
List	$p+r$	$m\text{-}r + \sum_{i=1}^{m} N(b_{l_i})$	$p+l$	$m\text{-}r + \sum_{i=1}^{m} N(b_{l_i})$
Structure	$p+r$	$m\text{-}r + \sum_{i=1}^{m} N(b'_{s_i})$	p	m

Table 7.1: The maximum and minimum cases of the indexing tree, where p is 0 if $m = 1$; otherwise, p is 1.

As an example, we evaluated the size of the indexing tree for predicate match/2 discussed in section 7.3. In this case, the number of clauses n is 12 and the number of partitions m is 4. Also, r, the maximum number of buckets which have more than one clauses, are defined as 1, 1, and 2 respectively for the constant, structure, and list data type. The other parameters are listed in Table 7.2. The result is shown in Table 7.3. According to the table, up to three choice points will be created for the execution of a predicate when the input argument is structure "sum/2".

i	$N(P_i)$	$N(b_{t_i})$	$N(b_{l_i})$	$N(b'_{c_i})$	$N(b'_{s_i})$
1	3	3	0	1	2
2	1	1	0	0	0
3	7	7	2	2	2
4	1	1	0	0	0

Table 7.2: Parameters evaluated for the example predicate

	Maximum tree		Minimum tree	
Input Argument	Number of cp-node	Maximum no. of t-nodes	Number of cp-nodes	Maximum of cp-nodes
Variable	$1 + 0 = 1$	12	$1 + 0 = 1$	12
Constant	$1 + 1 = 2$	$4 - 1 + 2 = 5$	$1 + 0 = 1$	4
List	$1 + 1 = 2$	$4 - 1 + 2 = 5$	$1 + 1 = 2$	$4 - 1 + 2 = 5$
Structure	$1 + 2 = 3$	$4 - 2 + 4 = 6$	$1 + 0 = 1$	4

Table 7.3: The example: the maximum and minimum size

7.4 Flat Indexing

The analysis of the WAM indexing shows that the WAM indexing has a harmful influence on the parallelism exposition because up to $m + 1$ choice points will be created when a predicate has m partitions. With a view to enhance the parallelism exposition, we proposed a new indexing scheme, which we will call *flat indexing*. This section presents the flat indexing scheme along with our analysis of its parallelism exposition.

7.4.1 Description of the flat indexing

In the WAM indexing scheme, a set of subsequent clauses with a variable key is defined as an independent partition. It is because a variable key is always unified with the input argument of any type. By doing this, the WAM can produce very compact code since each bucket always includes the clauses whose keys are of the same data type. However, as discussed earlier, one critical drawback of this approach is that more than one choice points are sometimes created in a search path.

The flat indexing scheme is based on the idea that all the clauses are looked upon as one partition and a specific bucket is chosen according to the type of the input argument. The key feature of the flat indexing scheme is that only a single choice point will be created for every predicate. For this, we have made the following modifications:

- A bucket is an ordered set of clauses whose key matches with the input argument or is a variable.
- A specific bucket, to be called a *failure* bucket, is introduced such that it is an ordered set of clauses whose key is a variable.

For example, among the twelve clauses making up predicate `match/2` in section 7.3, c_{10} and c_{11} have a list as their key, and clauses c_4 and c_{12} have a variable as their key. The bucket for the list is thus defined as $\{c_4, c_{10}, c_{11}, c_{12}\}$ and the failure bucket as $\{c_4, c_{12}\}$.

Using the new form of a bucket and the failure bucket, the flat indexing works as follows. As in the WAM indexing, a bucket is prepared for *list* data type, and a list of buckets are prepared for data types *constant* and *structure*. In the beginning of a code, *swtch_on_term* instruction dispatches the control to a destination. The semantics of the instruction is the same as the one in the WAM indexing except that the destination for each data type becomes the failure bucket when there exists no clause whose key matches with the data type. For *constant* or *list* data, instuctions *switch_on_constant* and *switch_on_strcuture* dispatch the control to the appropriate bucket depending on its data value. These instructions are the same as those in the WAM indexing except that they have an additional pointer to the failure bucket. If no bucket exists for the data value, the control is transfered to the failure bucket.

Figure 7.6 depicts the structure of code produced for a predicate by the flat indexing scheme. In principle, it is very similar to the code produced for a partition by the WAM indexing . However, as discussed previously, arguments of instructions *switch_on_term*, *switch_on_constant*, *switch_on_strucutre* are slightly different from those in the WAM indexing.

Instruction `switch_on_term` is slightly different from the one in the WAM indexing. When an input argument is not matched with the entry, in the WAM the control is dispatched to a failure service routine. However, in the flat indexing scheme, the control is dispatched to a failure bucket and then the clauses in the bucket will be executed. This is because the flat indexing has only one partition and any non-variable argument will be always matched with the variable key even if no clauses exist whose key matches with the argument.

predicate:	
	Switch_on_term *Start*, C_Switch | C_i | *Fail_bucket*, L_Bucket | C_i | *Fail_bucket*, S_Switch | C_i | *Fail_bucket*
Fail_bucket:	*code for* failure bucket
C_Switch:	Switch_on_constant [*pointers to buckets*,Fail_bucket] lists of buckets for constants
L_Bucket:	a bucket for lists
S_Switch	Switch_on_structure [*pointers to buckers*,Fail_bucket] lists of buckets for structures
Start:	Code for clasuses

Figure 7.6: A structure of the generated code

In order to clearly show how a predicate is coded by the flat indexing scheme, we provide in Figure 7.7 the structure of the code for predicate `match/2`.

7.4.2 Analysis of the flat indexing

As we did for the WAM indexing scheme, we have derived the number of the cp-nodes and the t-nodes in an indexing tree under the flat indexing scheme. The results are reported in Table 7.4. As shown in the table, the number of cp-nodes, that corresponds to the number of choice points, is always one.

```
              switch_on_term C1_Label, C_Swich, L_Bucket, S_Switch
C_Switch:     switch_on_constant 2, {a: C_a_Bucket, b: C_b_Bucket)

Fail_Bucket:  try C4_Code
              trust C12_Code
C_a_Bucket:   try C4_Code
              retry C5_Code
              retry C6_Code
              trust C12_Code
C_b_Bucket:   try C4_Code
              retry C7_Code
              trust C12_Code
L_Bucket:     try C4_Code
              retry C10_Code
              retry C11_Code
              trust C12_Code

S_Switch:     switch_on_structure 1, {sum/2 : S_sum_Bucket}
S_sum_Bucket: try C1_Code
              retry C2_Code
              retry C4_Code
              retry C8_Code
              retry C9_Code
              trust C12_Code

C1_Label:     try_me_else C2_Label
C1_code:      Code for match( sum(A,B), sum(C,D)) :-
                    match(sub(A+D-1), sum(C+B-1)) .
              ;
              ; Codes for C5 - C10
              ;
C11_Label:    retry_me_else  C11_Label
C11_Code:     Code for match( [a,A], [C,b] ) :-
                    match(a,C), match(A,b).

C12_Label:    trust_me_else_fail
C12_Code:     Code for match( X, numeric(Y) ) :-
                    match(numertic(X), numeric(Y))
```

Figure 7.7: An structure of the produced code for "match/2"

Input Argument	Number of created cp-node	Maximum number of t-node	Mininum number of t-nodes
Variable	1	n	n
Constant	1	$v + N(b'_c)$	v
List	1	$v + N(b_l)$	$v\ N(b_{l_1})$
Structure	1	$v + N(b'_s)$	v

Table 7.4: The analysis result: v is the number of clauses whose key is a variable and b'_c (*resp.* b'_s) is the bucket whose clause size is the largest among the buckets with a constant (*resp.* a structure).
key

Table 7.5 shows the values which we obtained by applying the analysis result in Table 7.4 to predicate `match/2`. It also shows the values taken from Table 7.3. From the table, we can clearly see the principal difference between the WAM indexing and the flat indexing; the number of choice points created for each predicate is always one in the flat indexing, while it can be three in the WAM indexing. Interpreted within the context of OR-parallelism, the reduction of choice points corresponds to the increase of the amount of OR-parallelism exposed for each choice point. In other words, the flat indexing contributes to the reduction of the non-leaf nodes the search tree, thereby, it increases the amount of OR-parallelism per node.

In addition, a close examination of the t-nodes in the table reveals a strange result. The size of the indexing tree is different in terms of the number of t-nodes. For example, when the input is a list, the number of t-nodes is five for the WAM indexing scheme and four for the flat indexing scheme. We will explain the reason by using their indexing trees created when the input argument is a list. Among the five t-nodes for the WAM indexing scheme, it is found that the first t-node is for the failure (case 1) resulting from *switch_on_term* instruction in the first partition and the remaining ones are for clause tries (case 2) respectively for c_4, c_{10}, c_{11}, and c_{12}. On the other hand, all the four nodes for the flat indexing scheme are for clauses tries. As a matter of fact, in the flat indexing scheme, all the terminal nodes are always for clause tries and their number is always smaller than or equal to the one in the WAM indexing scheme. Interpreted within the context of the parallel execution, the removal of terminal nodes caused by case 1 corresponds to the reduction of task switching by a scheduler. In general parallel logic programming systems, the task switching is a very expensive operation because the scheduler must prepare the environment for the destination node. When a scheduler task switches to a terminal node in case 1, it will finish the task right after the task switching, just wasting expensive system resource. Therefore, the reduction of terminal nodes caused by case 1 enhances the system performance by eliminating unnecessary context switching for the node.

	Maximum tree		Minimum tree	
Input Argument	Number of cp-node	Maximum no. of t-nodes	Number of cp-nodes	Maximum of t-nodes
Variable	1 (1)	12 (12)	1 (1)	12 (12)
Constant	1 (2)	5 (5)	1 (1)	2 (4)
List	1 (2)	4 (5)	1 (1)	4 (5)
Structure	1 (3)	6 (6)	1 (1)	2 (4)

Table 7.5: The maximum and minimum size for `match/2` under the flat indexing, where the numbers enclosed in parentheses are for the WAM indexing scheme.

7.5 Evaluation Results

In previous sections, we showed that the size of the indexing tree generated in the flat indexing scheme is always smaller than that in the WAM indexing. The reduced number of choice point creation is the primary benefit. With the absence of some terminal nodes caused by failure from *switch_on_term* (*_constant, _structure*), the total number of instructions executed for a benchmark will be also reduced. On the other hand, some more instructions are required in the implementation of the flat indexing scheme, particularly for implementing the bucket. Therefore, the code size may be larger than that in the WAM indexing. With respect to the above qualitative estimation, questions still remain:

- What fraction of Prolog programs benefit from the flat indexing?

- How much will the flat indexing affect the size of the indexing tree for the benchmarks which benefits from the flat indexing?
- How much will the flat indexing affect the size of the code?

As discussed earlier, the reduction of the indexing tree directly contributes to the enhancement of OR-parallelism per each node of the search tree as well as the removal of bogus task switching by a scheduler. Therefore, the first and second questions are to see how much effective the flat indexing will be for practical applications. On the other hand, compared with the WAM indexing, the flat indexing may generate less compact code. Therefore, the third question is to see how less compact the code will be under the flat indexing.

In order to answer the question, we conducted an experiment. The experiment is based on the TCWAM Prolog system. The normal TCWAM Prolog compiler produces the WAM code in which the indexing part is based on the flat indexing scheme. Linked with an emulation engine, the WAM code has been executed on a HP's SPP-IAX system. By modifying the TCWAM, we implemented another version that supports the WAM indexing scheme. This version will be called the TCWAM-NFI (non-flat indexing), We selected 17 benchmarks which have been frequently used in the evaluation of Prolog systems [6, 29]. Respectively for each version, we measured the following three performance criteria:

- the size of the indexing tree,
- the code size of assembly source, object, and executable code, and
- the execution time.

Table 7.6 shows the size of the indexing tree for each benchmark. Over 50 % of the 17 benchmarks benefit from the flat indexing. In the table, those are indicated by asterisks. Overall, 8 % more choice points have been created and 19 % more *switch_on_term, switch_on_constant*, or *switch_on_structure* failures occur under the WAM indexing. For the set of benchmarks which are affected by the flat indexing, 15 % more choice points are created and 35 % more *switch_on_term, switch_on_constant, or switch_on_structure* failures occur under the WAM indexing.

Table 7.7 shows the size of the code and execution time measured for the TCWAM system. Since the TCWAM translates Prolog code into C code via

	Flat indexing		Wam indexing		Comparison	
Prolog Program	cp-nodes number f1	t-nods number f2	cp-nodes number w1	t-nodes number w2	cp-node ratio w1/f1	t-node ratio (w2/f2)
boyer*	179476	89157	282097	194437	1.57	2.18
browse*	274714	271400	278387	281873	1.01	1.04
cal	30019	22641	30019	22641	1.00	1.00
chat_parser*	32620	39539	35845	40354	1.10	1.02
crypt	81	222	81	222	1.00	1.00
ham	359736	359734	359736	359734	1.00	1.00
meta_qsort*	2725	3598	2725	4405	1.00	1.22
nand*	8142	8566	8142	8665	1.00	1.01
nrev	580	578	580	578	1.00	1.00
poly_10*	14039	12531	18975	30733	1.35	2.45
queens10*	533231	533217	634592	634578	1.19	1.19
reducer*	10433	15986	11904	15986	1.14	1.00
sdda*	568	709	568	744	1.00	1.05
sendmore	12071	26128	12071	26128	1.00	1.00
tak	63625	15916	63625	15916	1.00	1.00
tak_gvar	790	418	790	418	1.00	1.00
zebra	14498	17315	14498	17315	1.00	1.00
average					1.08	1.19
average*					1.15	1.35

Table 7.6: Comparison of the indexing tree size

the Warren Abstract Machine, we have measured the assembly code size which obtained by "gcc -S -O2 program-name". We used "gcc" version 2.6.3 and the optimization level "-O2".

Prolog Program	Assembly code size KBytes	Object code size KBytes	Executable code size KBytes	Execution time msec
boyer*	283	63	266	1374
browse*	79	20	237	1662
cal	75	19	237	180
chat_parser*	794	182	356	333
crypt	59	15	237	13
ham	61	16	233	1875
meta_qsort*	71	18	238	21
nand*	431	95	299	60
nrev	41	11	233	277
poly_10*	71	18	238	109
queens10*	41	11	233	6018
reducer*	217	51	262	100
sdda*	141	34	250	6
sendmore	59	14	233	139
tak	20	6	229	298
tak_gvar	26	8	229	10
zebra	42	12	233	112

Table 7.7: The code size and execution time measured by the TCWAM (under the flat indexing)

Table 7.8 shows the ratio of the TCWAM-NFI to the TCWAM. On average, the assembly code size and object code size in the flat indexing are respectively 2 and 10 % larger than those in the WAM indexing, while the executable code sizes are the same. For the benchmarks which are affected by the flat indexing, the assembly code size and object code size in the flat indexing are respectively 5 and 10 % larger than those in the WAM indexing. This indicates that the flat indexing scheme does not lose much in terms of code compactness.

Prolog Program	Assembly code size	Object code size	Executable code size
boyer*	0.95	0.94	1.00
browse*	0.94	0.86	1.00
cal	0.99	0.95	1.12
chat_parser*	0.93	0.91	0.98
crypt	1.00	0.88	0.98
ham	1.00	0.89	1.00
meta_qsort*	0.94	0.84	1.00
nand*	0.97	0.95	1.00
nrev	1.00	0.92	1.00
poly_10*	0.97	0.90	1.00
queens10*	0.95	0.84	1.00
reducer*	0.95	0.92	1.00
sdda*	0.99	0.94	1.00
sendmore	1.00	0.94	1.00
tak	1.05	0.87	1.00
tak_gvar	1.04	0.90	1.00
zebra	1.02	0.93	1.00
average	0.98	0.90	1.00
average*	0.95	0.90	1.00

Table 7.8: Comparison: each entry is the rate of the TCWAM-NFI over the TCWAM (*i.e.*, TCWAM-NFI/TCWAM)

7.6 Synthesis

In this chpater, we have presented a new indexing technique, which we have called flat indexing, and its implementation. In the flat indexing, the number of choice points is not more than one for every invocation of each predicate. Therefore, a choice point represents all the available parallelism, *i.e.*, the number of available alternatives, in itself. By this, the parallelism can be more readily available in the parallel execution. Moreover, the bogus branches which will fail right after being taken has been removed in the flat indexing.

Our evaluation is based on a canonical representation of the execution tree called indexing tree. By comparing the indexing tree for a set of benchmarks, we have proven that the number of choice points and of bogus branches are reduced when the flat indexing is applied. The evaluation results show that one half of the benchmarks benefit from the flat indexing and that the number of choice points is reduced by 15 %. Moreover, the number of t-nodes is reduced by 35 %. We believe that the reduction will contribute to higher parallel performance due to the enhanced parallelism per node as well as to the reduction of task switching.

Chapter 8

Implementation and Performance Evaluation

In the previous chapters, we discussed our parallel execution model and its relevant implementation techniques. To verify the performance, we implemented the execution model on a distributed shared memory multiprocessor system. The implementation incudes a front-end compiler, that produces abstract machine code from a Prolog program, and a translator which generates from the abstract machine code a parallel-C code executable on the SPP-IAX system. It also includes a runtime scheduler which supports the following three scheduling algorithms: a top-most scheduling, a bottom-most scheduling, and a top-most scheduling with an architectural optimization. This chapter presents the experiment which ranges from an introduction of the target architecture to the performance evaluated for a set of benchmarks and its analysis.

8.1 Introduction

A parallel execution model of logic programs consists of a set of abstract specifications which illustrate the execution mechanism. Even though the specifications are clear and accurate, the implementation study is essential for the identification of implementation issues as well as the verification of the performance due to the following reasons:

- The interaction between the sequential engine and its associated extension for parallel support cannot be clearly defined in parallel execution models, because they are usually dependent upon the characteristics of the target sequential engine.

- There are a number of implementation issues which cannot be identified in the layer of the parallel execution model but have crucial impact on the performance because they are usually dependent upon the characteristics of the target parallel architecture.
- Prediction of the performance of parallel execution is not practically possible because the dynamic nature of logic programs prevents us from developing an appropriate performance model.

Upon the requirement, it is essential to make a prototypical implementation of our parallel execution model on a target parallel architecture. Although the prototype does not have to be complete in that it can support the complete list of Prolog library in commercial Prolog systems, it must be stable and robust enough to evaluate almost every benchmarks used in other research cites. This chapter presents the prototypical implementation of our parallel execution model and the result of the performance evaluation. The rest of the chapter is organized as follows. Section 8.2 presents the implementation of our parallel execution model. Section 8.3 presents the performance evaluated for a set of benchmarks and its analysis. Section 8.4 summarizes the chapter.

8.2 Overview of the Implementation

This section presents an overview of our target machine, the Exemplar system architecture, and discusses the details of the implementation.

8.2.1 Exemplar SPP IXA-0016

Exemplar SPP IXA-0016 is a multiprocessor system. The model we used has 16 CPUs, each of which is a PA-RISC 1.1, and runs under the SPP-IX 3.1 operating system. Figure 8.1 shows a conceptual diagram of the Exemplar system. The system is a distributed shared memory system with two tiers of memory latency. The hypernode crossbar constitutes the first tier, and the SCI rings constitute the second. Interhypernode accesses take longer than intrahypernode accesses but are transparent to the process because of using shared memory.

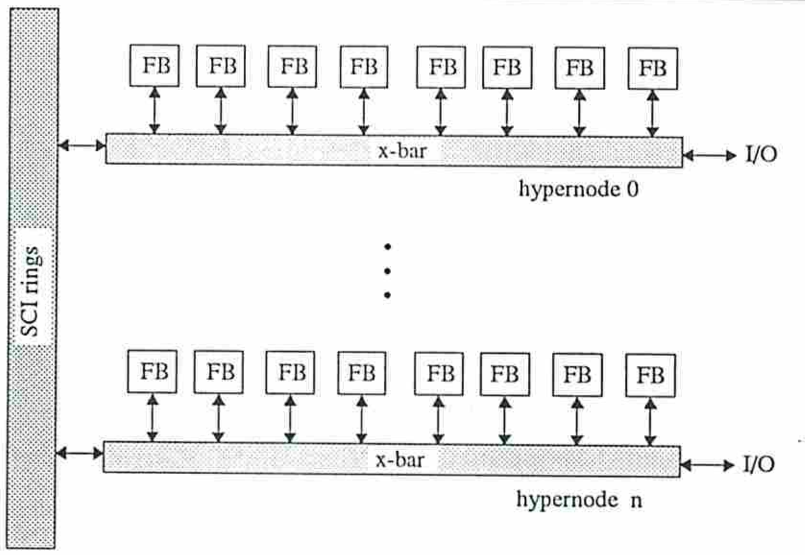


Figure 8.1: Conceptual overview of the Exemplar system

The Exemplar SPP-1200 system consists of 1 to 16 hypernodes as shown in Figure 8.2. Processors in a hypernode are arranged in functional blocks (FB), each containing two processors, 128 Mbytes to 512 Mbytes of memory, and some control devices. There is one memory unit in the block that holds hypernode-private memory data, global memory data, and network cache data. Four functional blocks constitute a hypernode. Functional blocks within a hypernode communicate with each other, with memory, and with peripherals via a 5 port non-blocking cross bar. Functional blocks communicate across hypernodes via four SCI rings.

CPUs communicate directly with their own instruction and data caches, which are 1 Mbyte in size and are located 1 clock away from the CPU. The functional block's two CPUs communicate with the rest of the machine through the CPU agent. The Convex Coherent Memory Controller (CCMC) provides the interface between the functional block's 2 memory banks and the rest of the machine. All intrahypernode memory accesses take 50 clocks, even though they can be fulfilled from the functional block's own memory block. This is because they must traverse the crossbar, which gives equal accesses to all hypernode memory from all functional blocks.

Each hypernode contains one or more hypernode-private memories that can be accessed from any CPU within the hypernode. Hypernode-private memory is not accessible from other hypernodes. Multiple hypernode-private memories operate independently and may be hardware-interleaved to provide greater bandwidth. Besides, each hypernode contains one or more global memory blocks. Global memory blocks provide global memory accessible by all hypernodes in the system, including the one containing it, a network interface used to connect to other hypernodes, and a network cache. The network cache en-caches all global memory data imported by this network interface from other hypernodes on the network.

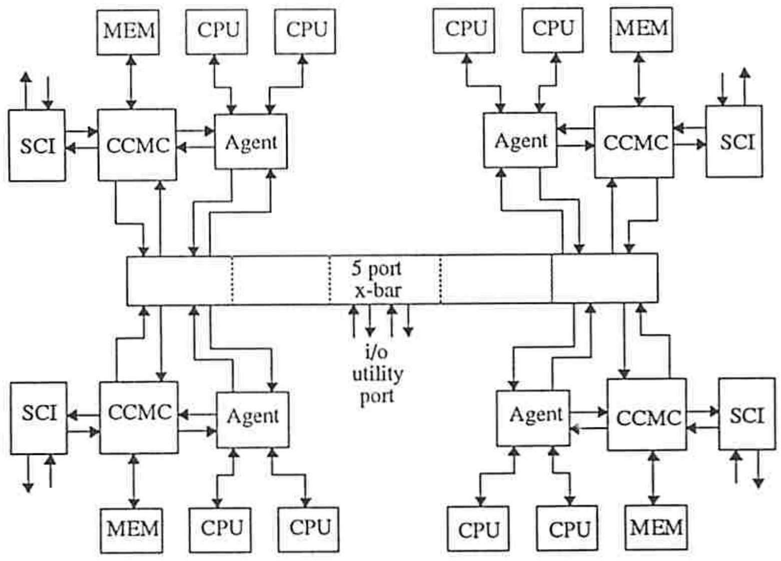


Figure 8.2: Exemplar hypernode architecture

The Exemplar system uses four SCI rings attached to each hypernode as the interconnection network between hypernodes, as shown in Figure 8.3. Four rings provide higher interconnection bandwidth, lower interhypernode latency, and redundancy in case of ring failure. Sequential memory references to global memory

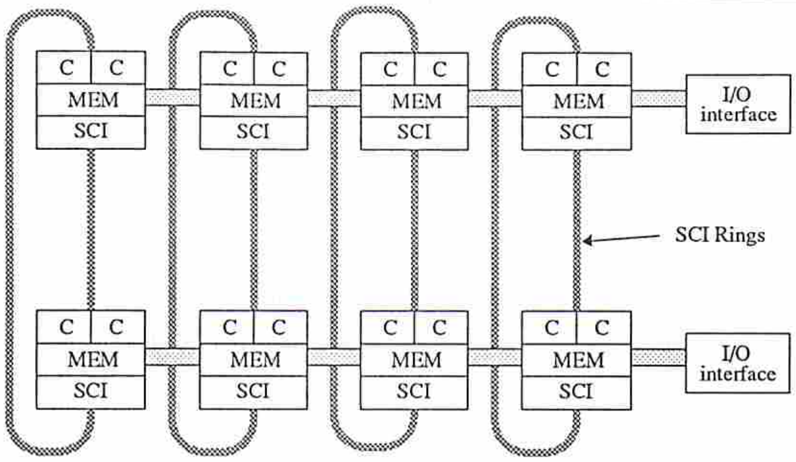


Figure 8.3: Exemplar SCI ring interconnect

are interleaved across the four rings. This is accomplished using the ring in the same functional unit as the target memory, because the memories are interleaved on a 64-byte basis. The four SCI network is interleaved on this basis as well; the network cache size is 64 bytes. This ring interleaving tends to balance the traffic across all four rings. (Global memory references from a CPU to the global memory on the same hypernode do not use the hypernode interconnect network and are not en-cached in the network cache.)

8.2.2 Prototypical implementation

Figure 8.4 shows a diagram of our experimental prototype built on an Exemplar SPP IXA-0016 system. The main components of the prototype are the front-end TCWAM compiler and a runtime system. This subsection presents the implementation.

The front-end TCWAM compiler takes an input Prolog program and produces the C code which consists of a sequence of C macros. The main features are as follows:

- Fast C translation scheme described in chapter 6 is used.

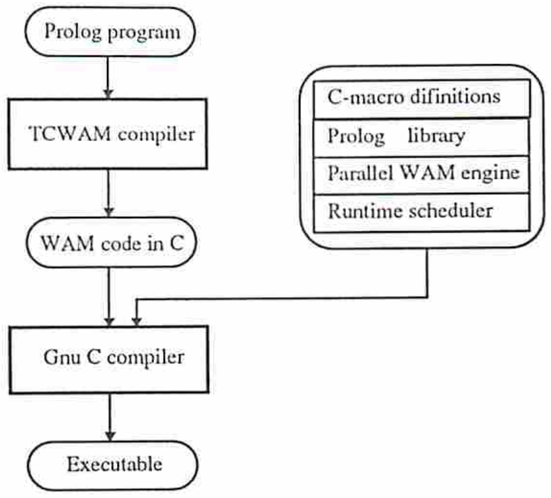


Figure 8.4: The overview of the prototype implementation

- The compiler supports multiple modules for modular program development.
- The compiler supports the ISO standard Prolog syntax.
- The compiler supports most built-in predicates which include the following functions.
    - Input and output of terms
    - Arithmetic operators
    - Term comparison
    - Constant processing
    - Term processing
    - Test predicates
    - Control
    - Modification of the program
    - All solutions

Linked with the runtime system, the C code produced by the front-end compiler is compiled into the executable. The runtime system consists of the following components:

- a C macro module,
- a TCWAM engine with some extension for parallel support, and
- a scheduler.

The macro module consists of a set of macros which support C code translation, define the extended TCWAM instruction set, and support the engine and the scheduler interfaces.

Parallel extension of the WAM

As noted earlier, the original WAM is extended in our prototypical implementation in support of the parallel model. The extension includes some additional instructions and registers to provide the following functions:

1. the flat indexing,

2. the manipulation of conditional variables, and

3. the interface of the engine and the scheduler.

Now that the instructions and its operational aspects of the flat indexing method has been discussed in chapter 7, this subsection is devoted to the discussion only on the second issue, while the third issue is remained to be discussed in the next subsection.

Regarding the manipulation of the conditional variables, the most important issue is how to identify the condition variables. To this end, we provide two instructions in association with the creation of conditional variables. We begin the discussion by presenting the definition of conditional variables.

Definition 8.2.1 (Conditional variables) *Given a clause variable, regardless whether it is a permanent or a temporary variable, the variable is defined as a conditional variable, if it does not get bound during unification.*

From the above definition, we can observe that the place in which a variable appears in the clause has to do with whether it is a conditional or a non-conditional variable, as summarized below.

Variable type	First Appearance	
	HEAD	GOAL
Temporary variable	(a) unify_x_vvariable	(b) put_x_variable unify_x_variable
Permanent variable	(c) unify_y_variable	(d) put_y_variable unify_y_variable

Table 8.1: WAM instructions used for the first appearance of variables

- If a variable does not appear in the head, it is always a conditional variable.
- Otherwise, it can be either a conditional or a non-conditional variable.

Variables in the first case are always subject to the initialization as conditional variables. As for variables in the second case, we can divide them into two classes as follows:

- Class A: If a variable is a singleton which appears in a head of a clause, it always gets bound in the unification regardless of the type of the input argument term.
- Class B: If a variable is a non-singleton, *i.e.*, it belongs to to a compound term in a head of a clause, it remains unbound only for some input argument. As an example, consider a head term of a clause, `head(f(X)`, If the input argument is `A`, variable X remains unbound because `f(X)` is bound to variable `A`. On the other hand, given an input argument `f(1)`, variable X is bound to `1`.

Variables in class B are subject to the initialization as conditional variables only when they remain unbound after unification. This indicates that we must examine variables in the class B after the unification to find out whether it is bound or not.

For the implementation of the initialization of conditional variables, we introduce two new instructions and modify some existing WAM instructions. The new instructions are as follows:

- Instruction "`check_y_cv V`" inspect variable `V` whether the variable is bound or not. If the variable is not bound, it is initialized as a conditional variable in the Tagged Binding Array.
- Instruction "`create_y_cv V`" initializes variable `V` as a conditional variable.

In the detection and initialization of a conditional variable, the compiler uses the variable's type and the place of its first appearance in the clause. Table 8.1 shows a summary of such WAM instructions associated with variables. It contains four sections, (a), (b), (c), and (d). They are different from one another in terms of the variable's type and the place of the first appearance. For example, section (a) contains the WAM instructions which will be used to compile temporary variables that appear in the head of a clause.

The compilation of each section is processed as follows.

- For each variable which will be compiled into the WAM instructions in (c), instruction `check_y_cv V` is produced just after the code for head unification.
- For each variable which will be compiled into the WAM instructions in (d), instruction `create_y_cv V` is produced just after the unification code.
- Variables which will be compiled into the instructions in (b) must initialized as conditional variables. But, we cannot use instruction `create_y_cv V` and `check_y_cv V` because they are temporary variables which use argument registers instead of regular environment slots. Therefore, the instructions in (b) are modified so that they are able to initialize their operand variables as conditional variables.
- For variables which will be compiled into the instructions in (a), it is the most efficient to create them as conditional variables only for those which are not bound after the unification, as does in the case of (c). However, it is not possible because instruction `unify_x_variable` in (a) appears as well in (b). Instead, we treat them as in the case of (b) just for reasons of the consistency.

In order to make the compilation process more clear, we provide an example Figure 8.5. The figure depicts the code for the head unification which results from the compilation.

8.2.3 Scheduler

Our system uses the WAM [78] as the basis of its sequential engine. It extends the WAM with some instructions and data structures in support of OR-parallelism. In the previous subsections, we discussed the extension related with the management of conditional variables. In this subsection, we discuss the part of the extension which is associated with runtime scheduling.

Private and public nodes

Choice points, called *nodes* when discussed in the context of a search tree, are classified into the following two types: private and public nodes. As a matter of fact, this has been used in other OR-parallel models [15]. The main difference between private and public nodes consists in the representation and management of available work. A private node belongs to only a single worker. Because only the worker can take alternatives from the node, the data structure and its management are in principle identical to those in the sequential WAM. A public node is shared by the workers in the system. Because more than one workers would take the alternatives, a node possesses a slightly modified representation of the data structure and its management. During execution, only a subset of the nodes in the search tree are made public because some operations, particularly backtracking, are expensive when carried out for public node. An in-depth discussion will be offered shortly.

Public node and the management of its alternatives

The extension of the data structure and instructions made for a public node consists in the representation of the alternative clauses.

Chapter 7 discussed the flat indexing which is a compilation method to produce code for indexing alternative clauses of a predicate. The code produced through

```
[Example clause]

randomize(In,[X|Out],Rand) :-
    length(In,Lin),
    Rand1 is (Rand * 17) mod 251,
    N is Rand1 mod Lin,
    split(N,In,X,In1),
    randomize(In1,Out,Rand1).

[A part of the generated code]

Begin Clause

      allocate(7)
      get_y_variable(4,0)  ; Unification code
      get_list(1)
      unify_y_variable(3)
      unify_y_variable(1)
      get_y_variable(6,2)
                           ; Creation of conditional variables
      check_y_cv(3)        ; 'X'
      check_y_cv(1)        ; 'Out'
      create_y_cv(0)       ; 'Rand1'
      create_y_cv(2)       ; 'In1'
      create_y_cv(5)       ; 'Lin'

      put_y_value(4,0)
      put_y_variable(5,1)
      call(length/2,2)
          *
          *
End clause
```

Figure 8.5: An example clause and its compilation: the clause is from benchmark "browse.pl". Variables 'X' and 'Out' belong to section (c) and 'Rand1', 'In1', and 'Lin' belong to section (d).

the flat indexing is efficient in terms of the size of the search tree. The flat indexing technique is implemented by means of the WAM indexing instructions, "try", "retry", and "trust". In a compiled code resulting from the flat indexing, the subset of alternative clauses chosen for some indexing key is expressed by "try", "retry", and "trust" instructions, providing it consists of more than one clauses.

As we pointed out earlier, the representation of alternatives in choice points for private nodes is the same with the one implemented in the sequential WAM. It is because only one worker handle the alternatives. On the other hand, the choice points for public nodes are represented differently mainly for reasons of the mutual exclusion among workers which attempt to take alternatives from the choice point.

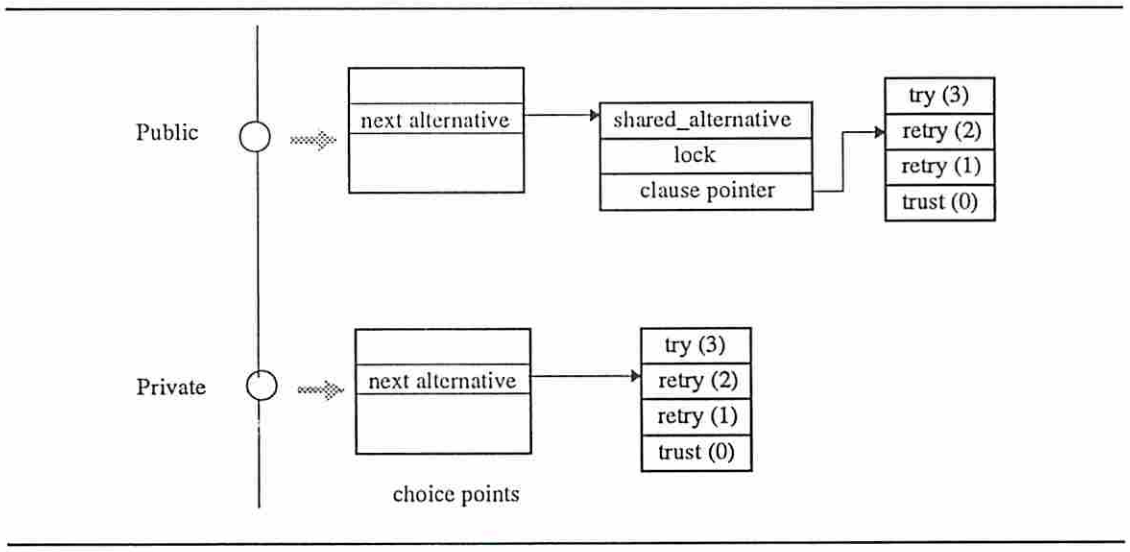


Figure 8.6: Choice points and alternative pointers

Figure 8.6 shows a diagram which depicts the representation of alternatives in choice points. It shows that the *next alternative field* in the choice point for a private node keeps the address of the next alternative clause as does in the sequential WAM. In the choice point for a private node, the next alternative field contains a pointer to an auxiliary data structure which consists of three fields: a special instruction `shared_alternative`, a variable to be used as a lock, and an *alternative clause pointer*.

When a worker attempts to take an alternative from a public node, it executes automatically instruction `shared_alternative`. The instruction takes the next location as its operand and uses it as the lock variable for maintaining the mutual exclusion among multiple workers which attempt to access the alternative clause pointer. If the content of the variable indicates that the next alternative pointer is not being accessed by any worker, the instruction sets the lock variable to an appropriate value to prevent other workers from accessing the alternative clause pointer.

Once a worker is allowed to access the next clause pointer in the auxiliary data structure, the control of the worker is transferred to "try", "retry", or "trust" instruction depending on the content. In the beginning of the execution, the instruction sets the next clause pointer to the address of the next alternative clause and resets the lock to allow other workers to access the alternative field in the choice point.

Management of load information

From the viewpoint of load balancing, when a worker becomes idle, it is generally the most efficient for the worker to share some tasks with the most heavily loaded worker. For this, the parallel system must provide an accurate measure for the available work. Moreover, it must be able to identify the accurate amount of the avaliable work which keeps changing under dynamic scheduling. The amount of available work associated with a worker is usually called *load* and in most parallel implementation the number of unexplored branches has been used as its measure.

In our system, the load is divided into two classes: *public* and *private*, according as the search tree is divided into a public and a private part. The public load refers to the number of unexplored branches in the public choice points. The private load refers to the number of unexplored branches in private choice points. The amount of public load associated with a worker is influenced by the scheduling activities of the other workers. However, the amount of private load is not affected by the others.

It is prone to cause large overhead to trace the private load upon necessity during execution. To avoid the overhead, the current implementation provides a global register for each worker which contains the amount of the private load. Besides, "try", "retry", and "trust" instructions are extended to have an additional operand which is the number of alternatives following the instructions. It is shown in Figure 8.6. Each choice point has an additional field, `parent_private_node`, to have the summation of the private load for all the ancestor private nodes. When a node is created, the summation of `parent_private_node` in the parent node and the number of unexplored branches in the parent node is made as the value of its `parent_private_load`. In consequence, the current private load amounts to the summation of `parent_private_load` of the current node and the number of unexplored branches in the current node and it is maintained in the global load register.

8.2.4 Runtime scheduler: Top-most Scheduling

In our system, the scheduler is implemented on top of the common ancestor based representation of the search tree discussed in chapter 5. This subsection presents a brief description of the runtime scheduler.

The procedure of a task search

Whenever a worker exhausts its private work, it looks for some avaliable work. Subject to the scheduling policy and its associated data structures, a task search refers to finding a node which has at least one unexplored branches. Once a worker succeeds in locating a node, which we will call the *destination* node, it performs task switching from its standing node to the destination node.

In the procedure (Figure 8.7), the worker attempts to search the public ancestor nodes for nodes which haves any available task. When more than one nodes are found in the public nodes, the top-most one is selected as the destination node. If no such nodes are found, the worker tries to find some public work from workers which lies relatively in the upper part of the search tree. If the worker fails to find any work from them, it also tries to get some public work from workers which resize relatively in the lower part of the search tree. If it fails again to find available work, it selects the worker which has the largest private work and asks the worker to publicate some of the private work.

The procedure of executing cut

"Cut" is a built-in predicate. It puts a side effect on the procedure of the backtracking usually to prune away some branches which do not contribute to the solution or are redundant. The current implementation of "cut" is based on the approaches which have been shown effective in some other implementations [8].

Concerning the implementation of "cut", it should be noted that the execution of "cut" is subject to pruning operations of other "cut". Hence, the pruning to be made for some public node which belongs to the scope of a "cut' is possible only when the current branch executing the "cut" is the leftmost one for the node. Figure 8.9 shows an example. The figure depicts the situation in which a cut is encountered in branch b_5. Assume that the *scope node* of the "cut" be n_2, where the scope node refers to the node created for the predicate which contains the "cut". In this situation, branch b_2, b_3, and b_4 are in the scope of the "cut" and subject to pruning. As b_5 is in the left of b_5, we can prune away branch b_4. However, as b_2 and b_3 are affected by the "cut" only if branch b_7 does not contain another cut, the processing of the pruning must be either (1) postponed until it becomes the leftmost or (2) delivered to another left branch.

Our implementation of "cut" chooses the second option just for reasons of efficiency. Figure 8.10 shows the algorithm, in which BC corresponds to the register which contains the scope node of a cut. The information on the delayed cut is stored in the public frame which is allocated for each public node. The function `cut_prune()` is called inside a cut instruction and its arguments are respectively the name of worker which executes the cut and the scope of the cut. The function `delegate_cut()` is called whenever a worker backtracks and attempts to take a branch from a public node and its arguments are respectively the name of the worker and the node.

```
task_search(w, cp)
  worker w;
  choice point cp;
{
  choice point wcp;i

  for (wcp=cp ; ; ) {
    if ( public work found in the current choice point in wcp ) {
      take a branch;
      return;
    }
    wcp = find_public_work (w, wcp);
    if ( wcp is not null ) /* public work is found */
      make_up_worker(w, wcp)
      take a branch;
      return;
    }
    else if ( (wcp=worker_with_work(w, wcp, ABOVE)) != NULL or
              (wcp=worker_with_work(w, wcp, BELOW)) != NULL ) {
      make_move(w,wcp)
      return;
    }
    else if ( (wcp=publish()) != NULL ) {
      make_move(w,wcp);
      return;
    }
  }
}
```

Figure 8.7: The procedure of a task search

```
worker_with_work (w, flag)
   worker w;
   int flag;
{
   choice point cp;
   workers wl[worker_num];

   for (i = 0; i<worker_num; i++) {
     switch (flag) {
       case ABOVE:
         if ( cant(w->name,i) < w->cp->cp_depth) {
           if ( (wcp = find_public_work (w, wcp)) != NULL )
     return wcp;
         }
       case BELOW:
         if ( cant(w->name,i) < w->cp->cp_depth) {
           if ( (wcp = find_public_work (w, wcp)) != NULL )
     return wcp;
         }
     }
   return NULL;
}
```

Figure 8.8: The finding public task from other workers

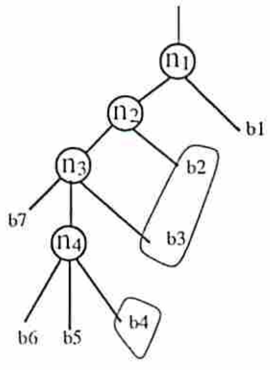


Figure 8.9: Exemplar hypernode architecture

```
cut_prune(w, BC)
{
  if ( BC->cp_depth >= w->private_top ) { /* private node */
    sequential_cut(BC);
    return;
  sequential_cut(w->private_top);
  for (cp := parent of w->private_top; cp< BC; cp := parent of n) {
    if ( w is not the leftmost ) {
       cp->pframe->cut_branch = my branch;
       cp->pframe->delayed_cut = BC;
       cp->pframe->delegate_cut = the live left branch;
       break;
    }
    sequential_cut(cp);
    kill workers between BC and cp;
  }
}

delated_cut(w, cp) {
  if ( cp->pframe->delegate_cut == my branch ) {
    if (any live left branch) {
       cp->pframe->delegate_cut;
       return;
    }
    scope = top->delated_cut;
    n = parent of cp;
    while (n->cp_depth <= scope) {
      if (w is not the leftmost) {
         n->pframe->cut_branch = my branch;
         n->pframe->delayed_cut = scope;
         n->pframe->delegate_cut = the live left branch;
         break;
      }
      sequential_cut(n);
      kill workers in the right of cp->pframe->cut_branch;
    }
  }
}
```

Figure 8.10: The algorithm of "cut"

8.3 Performance Evaluation

The main objectives of this performance evaluation are to identify the performance, to verify the underlying philosophy, and to validate the potential of the execution model. The performance is identified through practical execution of a set of prominent benchmarks which have been widely used because their parallelism and execution behaviors are clearly understood. The underlying philosophy is verified through the comparison and analysis of the performance under the three scheduling policies which have different characteristics in terms of the scheduling algorithm and scheduling optimization. Finally, the potential of the execution model is validated by the comparison of the performance of other systems.

The rest of section is organized as follows. In subsection 8.3.1, we describe the benchmark programs. In subsection 8.3.2, we analyze the average performance of benchmark set I obtained respectively schedule A, B, and C. In subsection 8.3.3, we analyze the performance of benchmark set I obtained by schedule A. In subsection 8.3.4, we analyze the performance of benchmark set II obtained respectively by schedule A, B, and C.

8.3.1 Description of benchmarks

The benchmarks used in the experiments are divided into two sets: set I and set II. Benchmarks in set I are relatively well understood with their granularity and execution behavior. It includes `8-queens1` (queens1_8.pl), `8-queens2` (queens2_8.pl), `tina` (tina.pl), `salt-mustard` (sm.pl), `parse2` (parse2.pl), `parse4` (parse4.pl), `parse5` (parse5.pl), `db4` (db4.pl), `db5` (db5.pl), `house` (house.pl), `parse1` (parse1.pl), `parse3` (parse3.pl), and `farmer` (farmer.pl). In this set, all the benchmarks except `8-queens1`, `8-queens2`, `tina` and `salt-mustard` have a repetition by some numbers. To indicate the repetition, the numbers are specified in their names. For example, benchmark parse4_5.pl is a program which repeats parse4.pl by five times.

Benchmark set I is used very often in many parallel logic programming systems [5, 15]. As a matter of fact, as the benchmarks have either no or little speculative work, they are highly suitable for the performance evaluation of non-speculative computation.

Benchmark set II consists of three programs `8-queens`, `zebra`, and `chat_parser`. These benchmarks are chosen to evaluate the performance of the computation which includes speculative work to some degree. Note that the `8-queens` program in this set is a version implementing a different algorithm from those in the first set.

8.3.2 The performance comparison of schedule A, B, and C

The current implementation has the following three different modes of scheduling.

- Schedule A: top-most scheduling with architectural optimization in which the scheduler usually takes the top-most work among the available work, while it sometimes chooses the nearest task to the worker.
- Schedule B: pure top-most scheduling in which the scheduler always takes the top-most task among the available tasks.
- Schedule C: bottom-most scheduling in which the scheduler always takes the bottom-most task among the available tasks.

The schedule B and schedule C enable us to evaluate the performance of the algorithmic optimization, while schedule A and schedule B enable us to evaluate the performance of architectural optimizations.

In this section, the performance of the system under the scheduling mode A, B and C is compared with the following three performance parameters: speedup, system utilization, and granularity of tasks. The performance comparison and analysis are based on average values calculated for all the first set of benchmarks. Throughout the section, an average refers to an arithmetic mean. Although the performance comparison based on the mean can never be complete, we believe that the mean value provides a very reasonable way. The reason for this is that the benchmarks in the first set have granularity balanced in terms of their size and have little or no speculative computation.

The performance comparison and analysis are aimed at identifying and showing how much our execution model is efficient and flexible in supporting algorithmic and architectural optimizations. On the algorithm side, we will show that a wide range of scheduling policies can be efficiently implemented in our execution model with a very small variance in scheduling overhead between them. On the side of architectural optimizations, the optimization in task allocation to archive higher locality in memory accesses is efficiently implemented in our model with minimal overhead during the process of locating an appropriate worker.

In this subsection, we first discuss the speedup which is one of the most important performance parameters within parallel computing. We then provide the analysis result on the system utilization and the task granularity and discuss the effect which such parameters have on the speedup. Finally, the performance comparison is made with other systems developed under the same rationale as the one under which our model is developed, *i.e.*, constant-time variable accesses.

Average speedup

Table 8.2, 8.3, and 8.4 show the results of the speedups respectively for schedule A, schedule B, and schedule C. Each table contains the speedups of all benchmarks in benchmark set I, in which each row is the speedup of a benchmark. Each column in the table is one instance of system configurations. For a benchmark, the speedup is measured for seven different configurations, which contain respectively 1, 2, 4, 6 8, 10, and 12 workers. The average speedup of all the benchmark is listed in the last row for each configuration. (Note that the execution time corresponding for these tables is found in the appendix A.)

According to Table 8.2, 8.3, and 8.4, the speedup values are quite different depending on benchmarks. In the configuration of 12 workers, the highest values reach about 10, while the lowest ones are about 1.5. The characteristics of benchmarks such as the amount of parallelism has a dominant influence on the magnitude of the speedup over the system characteristics such as scheduling policies. The degree of influence which scheduling policies has on the speedup differs from each other depending on benchmarks.

Prolog Program	Number of workers						
	1	2	4	6	8	10	12
queens1_8	1.00	1.95	3.75	5.55	7.03	8.52	9.47
queens2_8	1.00	1.96	3.73	5.61	7.34	8.78	10.39
tina	1.00	1.87	3.60	5.25	6.95	8.24	9.46
sm	1.00	1.91	3.57	5.04	6.56	6.92	7.35
parse2_20	1.00	1.88	3.47	4.96	5.40	6.06	7.29
parse4_5	1.00	1.90	3.56	5.15	5.09	6.76	7.43
parse5	1.00	1.90	3.68	5.26	6.71	7.62	8.21
db4_10	1.00	1.92	3.48	4.38	5.20	5.32	6.24
db5_10	1.00	1.92	3.59	5.05	5.33	7.09	6.02
house_20	1.00	1.95	3.56	5.27	6.73	7.95	9.08
parse1_20	1.00	1.77	2.48	3.17	3.46	2.79	3.17
parse3_20	1.00	1.78	2.19	2.92	3.54	3.00	3.43
farmer_100	1.00	1.72	1.72	2.24	1.97	2.36	2.69
average	1.00	1.88	3.26	4.60	5.49	6.26	6.94

Table 8.2: Speedup of benchmark set I: Schedule A

Prolog Program	Number of workers						
	1	2	4	6	8	10	12
queens1_8	1.00	1.95	3.63	5.11	6.54	7.47	8.57
queens2_8	1.00	1.96	3.68	5.35	6.81	8.05	8.94
tina	1.00	1.87	3.49	4.85	5.97	7.03	7.83
sm	1.00	1.92	3.54	4.73	5.89	6.42	7.39
parse2_20	1.00	1.88	3.03	3.88	4.15	4.77	4.94
parse4_5	1.00	1.90	3.50	4.86	5.09	5.50	6.78
parse5	1.00	1.90	3.58	5.05	6.35	7.37	8.54
db4_10	1.00	1.92	3.43	4.66	5.85	5.20	6.18
db5_10	1.00	1.92	3.44	4.88	5.41	5.30	6.02
house_20	1.00	1.94	3.60	4.99	5.94	7.08	8.15
parse1_20	1.00	1.74	1.92	1.89	2.20	2.44	2.64
parse3_20	1.00	1.76	1.95	1.84	2.24	2.48	2.47
farmer_100	1.00	1.66	1.67	1.52	1.72	1.84	1.89
average	1.00	1.87	3.11	4.12	4.94	5.46	6.18

Table 8.3: Speedup of benchmark set I: Schedule B

Prolog Program	Number of workers						
	1	2	4	6	8	10	12
queens1_8	1.00	1.96	3.84	5.42	6.76	7.85	8.56
queens2_8	1.00	1.98	3.83	5.46	6.92	8.25	9.19
tina	1.00	1.97	3.69	5.13	6.26	7.13	7.02
sm	1.00	1.98	3.64	4.93	6.30	5.20	7.34
parse2_20	1.00	1.91	2.52	3.17	3.83	3.27	3.35
parse4_5	1.00	1.97	3.50	4.27	4.55	5.31	5.68
parse5	1.00	1.98	3.77	5.34	6.60	7.76	8.80
db4_10	1.00	1.93	3.45	4.17	4.89	4.36	5.13
db5_10	1.00	1.95	3.45	4.48	4.45	4.57	5.27
house_20	1.00	1.96	3.56	4.85	4.61	4.35	4.80
parse1_20	1.00	1.49	1.43	1.58	1.84	1.85	1.89
parse3_20	1.00	1.40	1.40	1.50	1.76	1.77	1.86
farmer_100	1.00	1.37	1.11	1.40	1.38	1.65	1.79
average	1.00	1.83	3.01	3.98	4.63	4.87	5.44

Table 8.4: Speedup of benchmark set I: Schedule C

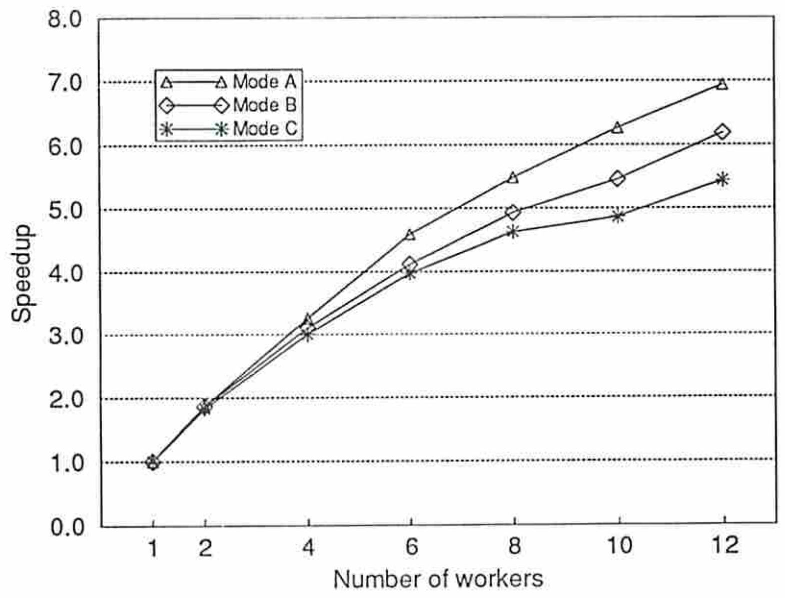


Figure 8.11: Average speedup of benchmark set I: schedule A, B, and C

Figure 8.11 shows the average speedup of all the benchmarks in benchmark set I. It contains three speedup curves in which each corresponds respectively to schedule A, B, and C. Schedule A provides the highest speedups for all the configurations over the other two. Between schedule B and schedule C, schedule B produces the higher speedup.

The growth rate of speedup over the number of workers decreases as the number of workers becomes larger. This results from a number of factors such as task scheduling overhead. To some degree, it reflects the architectural features of the target architecture. The SPP IAX-0016 is a distributed memory system. As the number of processor becomes larger, the remote memory access would take longer. Therefore, the growth rate can never be maintained the same as the size of the system become larger. Comparison between schedule A and schedule B provides a clear explanation. The two scheduling policies results in similar execution behavior in terms of task granularity because they are based on the top-most scheduling. The only difference between the two schedules is that schedule A includes an optimization which reduces the total amount of remote memory accesses. The result reported in Figure 8.11 shows that schedule A produces the higher growth rate over schedule B.

The performance gain of schedule A over schedule B results from the optimization made for architectural feature, particularly the reduction in the amount of remote accesses. The performance gain of schedule B over schedule C results from the optimization made for scheduling algorithm, particularly for larger task granularity. In the setting of 12 workers, the performance gains amount to the increase of speedups respectively by 0.76 and 0.74. These performance gains indicate that the optimizations are effective in both cases. On the other hand, the performance gain of schedule A over schedule C reflects the combined effect with respect to the scheduling algorithmic and architectural optimization. The results shows that the two types of optimizations are almost orthogonal. Therefore, the performance gain of the combined case amounts almost to the summation of the performance gains which can be obtained separately from each type.

Besides, Figure 8.11 shows that the difference of speedup among three schedule A, B, and C becomes bigger as the number of workers becomes larger in the system configuration. For example, the difference is about 0.1 under 4 workers and it becomes about 0.7 under 12 workers. It provides a proof that the optimizations both for the scheduling algorithm and architectural features are important for large scale parallel machines.

System utilization

Workers in parallel Prolog systems need to compute the code which will be computed in the sequential implementation. Besides, the schedulers need to execute some code associated with scheduling. As a matter of fact, the schedulers is embedded in workers and invoked by the workers upon necessity for such activities as finding available work and switching a task. Sometime, workers remain idle due to the lack of parallelism. Therefore, the system utilization will be discussed differently with respect to the following three classes of system time: *prolog time*, *task switching time* and *idle.*

- Prolog time refers to the execution time spent by a worker in executing the part of code which will be executed by the sequential system.
- Task switching time refers to the time spent by a scheduler in locating a task and preparing the environment for the task before executing the task.
- Idle time refers to the time in which a worker remains idle.

The measurement of prolog time is done by the instrumental code inserted in the entry and exit point of a prolog engine. The entry point refers to the location of the code which starts executing a new task and the exit point refers to the location of the code which fails to find a work through public backtracking, thus invoking a scheduling code to find a work. The task switching time is measured (i) in an entry and exit of functions for finding a public work and (ii) in an entry and exit of functions which prepare the environment. The idle time is not measured and is just calculated from the execution time minus the summation of the prolog and task switching time.

It should be noted that the time is measured by some instrumental code inserted in the code. Clearly, it is not as accurate as the values obtained through hardware support. When the instrumental code is inserted, the execution time

Prolog Program	Number of workers											
	2		4		6		8		10		12	
	P	T	P	T	P	T	P	T	P	T	P	T
queens1_8	99	1	96	2	95	3	92	4	90	5	83	8
queens2_8	100	0	96	2	96	2	95	2	92	4	91	5
tina	99	1	96	2	94	3	94	3	90	4	87	6
sm	98	1	92	4	87	6	85	7	73	12	65	12
parse2_20	98	1	91	4	87	7	72	8	65	9	65	9
parse4_5	99	1	93	4	90	5	67	8	73	8	67	8
parse5	99	1	96	2	92	4	89	6	82	6	74	8
db4_10	98	1	90	5	75	9	67	10	56	11	55	11
db5_10	98	1	92	4	87	6	69	10	75	9	53	13
house_20	98	1	92	4	91	5	87	6	83	8	79	10
parse1_20	92	4	65	9	56	12	46	15	30	14	29	14
parse3_20	92	4	58	11	51	12	47	14	32	12	31	13
farmer_100	88	7	45	16	39	20	26	17	25	17	24	18
average	96	1	84	5	80	7	72	8	66	9	61	10

Table 8.5: Prolog rate versus task switching rate (%) of benchmark set I: Schedule A

increases on average by 4 to 5 percents. The summation of utilization rate for all the three activities (prolog rate, task switching rate, idle rate) becomes about 95 percents. In order to make the utilization rate more understandable, the values are interpolated to be 100 percents.

Respectively for schedule A, B, and C, the prolog time and task switching time measured for each benchmark are listed in Table 8.3.2, 8.6, and 8.7. The last row of each table contains the average task switch time and prolog time for all the benchmarks calculated respectively in each of the seven configurations as did for the speedup. As shown in the tables. In the setting of 12 workers and scheduler A, the prolog rate ranges from 18 percents (farmer_100) to 88 percents (queens2_8). The prolog time of a benchmark reflects the task granularity and the nature of the inherent parallelism of the benchmark. The higher value usually indicates the higher speedup.

Prolog Program	Number of workers											
	2		4		6		8		10		12	
	P	T	P	T	P	T	P	T	P	T	P	T
queens1_8	99	0	96	2	94	3	94	3	90	5	88	6
queens2_8	100	0	97	1	97	2	97	2	94	3	91	4
tina	99	0	97	1	95	3	92	4	93	3	90	5
sm	99	1	93	3	86	6	84	7	76	8	76	10
parse2_20	98	1	82	6	74	7	61	9	60	10	54	10
parse4_5	99	1	94	3	92	4	75	6	69	7	73	7
parse5	99	1	96	2	96	2	94	3	92	4	92	4
db4_10	98	1	91	4	86	7	84	8	62	10	64	9
db5_10	98	1	91	4	90	5	77	8	63	9	62	10
house_20	98	1	94	3	89	5	84	8	82	9	80	9
parse1_20	91	4	52	12	36	11	33	12	31	13	29	15
parse3_20	91	3	53	10	35	11	33	12	31	13	27	14
farmer_100	85	6	44	15	28	15	25	17	22	19	20	19
average	96	1	83	5	76	6	71	7	66	8	65	9

Table 8.6: Prolog rate versus task switching rate (%) of benchmark set I: Schedule B

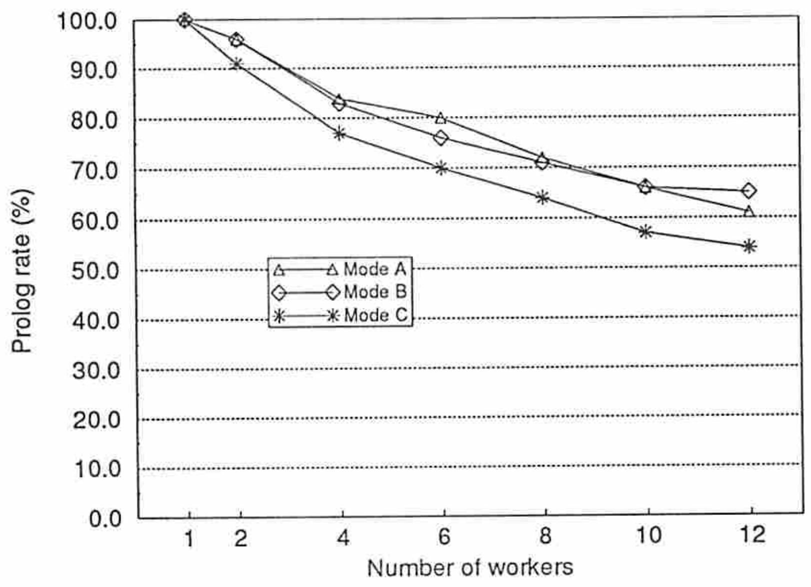


Figure 8.12: Average prolog rate of benchmark set I: schedule A, B, and C

Prolog Program	Number of workers											
	2		4		6		8		10		12	
	P	T	P	T	P	T	P	T	P	T	P	T
queens1_8	98	1	97	1	95	3	92	4	90	5	85	7
queens2_8	99	1	97	2	96	2	94	3	94	4	89	6
tina	98	1	96	2	94	3	91	5	89	6	76	6
sm	99	0	93	4	86	7	85	7	60	11	72	10
parse2_20	96	2	65	7	57	10	54	11	39	10	35	12
parse4_5	99	1	90	5	76	6	64	8	63	8	59	8
parse5	99	0	97	1	95	2	92	4	92	4	91	5
db4_10	96	2	88	6	73	7	67	8	50	12	51	12
db5_10	98	1	88	6	79	7	61	9	53	11	52	11
house_20	98	1	89	6	84	9	62	11	49	14	46	16
parse1_20	75	6	37	11	29	14	26	16	22	18	20	19
parse3_20	70	6	36	11	27	15	25	16	21	18	20	19
farmer_100	69	10	28	15	25	17	19	19	19	19	18	19
average	91	2	77	5	70	7	64	9	57	10	54	11

Table 8.7: Prolog rate versus task switching rate (%) of benchmark set I: Schedule C

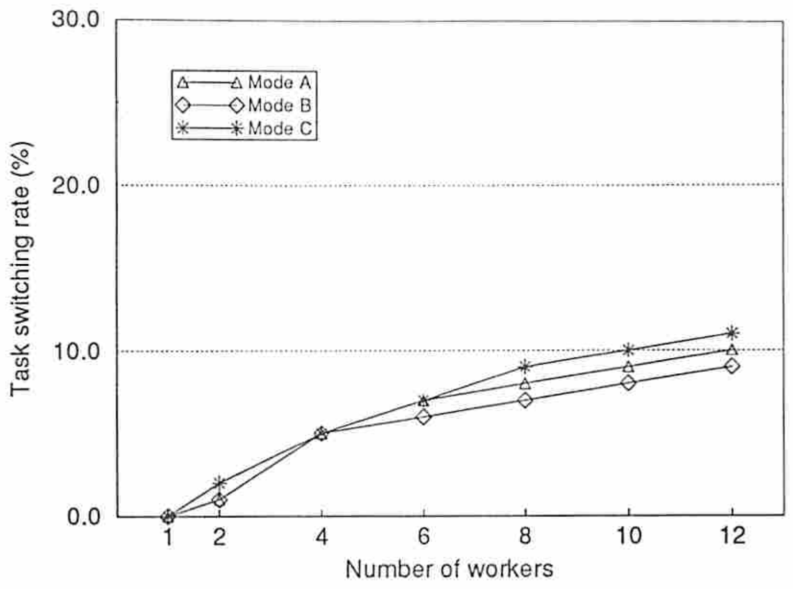


Figure 8.13: Average task switching rate of benchmark set I: schedule A, B, and C

The average values of the prolog rate, the task switching rate, and the idle rate are depicted in Figure 8.12, 8.13, and 8.14. Each figure contains three curves respectively for schedule A, B and C.

According to Figure 8.12, the average prolog rate decreases as the number of workers increases. Indeed, the average prolog rate in a single worker system is 100 percents because no task switching occurs, whereas it becomes about 60 percents in the configuration of 12 workers. The decrease is mainly due to the increase of task switching as well as some idle time which occurs due to the lack of parallelism. The prolog rate of schedule A and schedule B is higher than that of schedule C. It is mainly due to difference of the granularity. The prolog rate is almost the same between schedule A and B. In the discussion which follows shortly, it will be shown that the task switching of schedule A is higher than that of schedule B. It means that the prolog rate of schedule B must be smaller than that of schedule A. However, due to the architectural optimization, execution time of schedule A becomes smaller than the one of schedule B and consequently the prolog rate becomes relatively larger in the total execution time.

As opposed to the average, the average task switching rate increases as the size of the system becomes larger (Figure 8.13). On average, the task switching

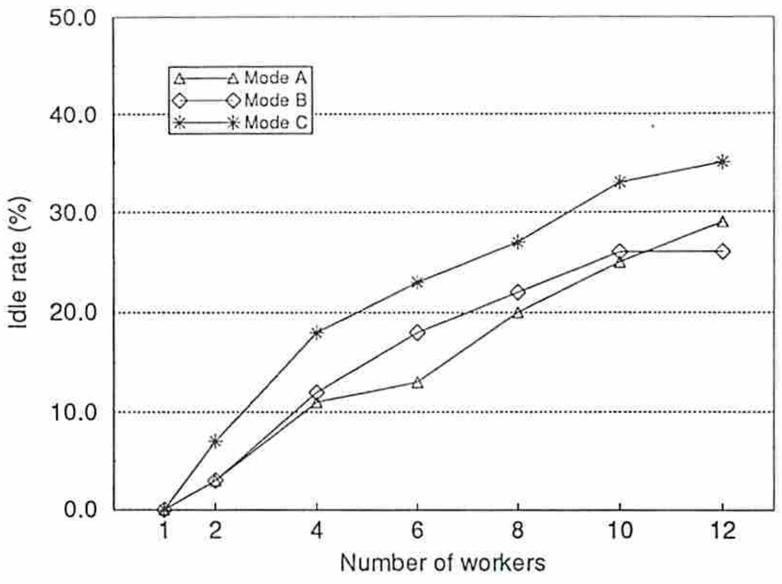


Figure 8.14: Average idle rate of benchmark set I: schedule A, B, and C

rate is about 10 percent of the total execution time in the configuration of 12 workers. The task switching rate of schedule C is higher than those of schedule A and B due to the smaller granularity. Compared with schedule B, schedule A has a higher task switching rate. It is because the architectural optimization may cause the selection of younger nodes which are not chosen in schedule A; since the younger nodes result in usually smaller granularity than the older nodes does, the granularity of tasks of schedule A is relatively smaller than those of schedule B.

The idle rate increase as the number of workers becomes larger (Figure 8.14). On average, the idle rate of schedule C is the larger than those of schedule A and C. It should be noted that the idle time in the figure is not exactly the time spent without any other work. It may include the time spent in some activities by scheduler other than the task switching and environment preparation. For example, it may include the time spent by a worker waiting for another worker to publish available work. This case occurs when another worker has some private work that can be shared with the idle worker. For this reason, the idle time becomes to some degree larger as the number of publications becomes lager. However, as the size of the system becomes larger, the increase rate of the idle rate becomes higher than that of task switching rate, because most of idle time is due to the lack of parallelism.

Computation efficiency

Workers in the parallel execution involve with some computation which cause parallel overhead. For example, task switching belong to such computation. Besides, they sometimes remain idle, if they fail to find some available work. These indicate that among the three classes of execution time, only the prolog time directly contributes to the speedup.

Let us define the prolog rate U_p as the rate of prolog time over the total execution time and then consider a system with N workers. Assuming that the instruction execution rate of the system is the same with that of the single worker system, the maximum possible speedup S_I becomes

$$S_I = N * U_p.$$

For example, in single worker system, the value of S_I is 1 because the instruction execution rate is 1 and the prolog rate is 1 (100 percents). In a system with more than one workers, the instruction execution rate is usually smaller than 1 mainly because of the latency of remote memory accesses. As a matter of fact, the instruction execution rate indicates how fast a worker executes an instruction. For a given system with N workers, in order to identify the effectiveness of architectural optimization implemented in schedule A, we compute the instruction execution rate within the prolog time respectively for schedule A, B, and C.

Let us first define computation efficiency E_c as the rate of the speedup S with respect to the ideal speedup S_I. The computation efficiency in a system with N workers is expressed as follows:

$$E_c = \frac{S}{S_I} = \frac{S}{(N * U_p)}$$

The computation efficiency E_c provides a measure to evaluate the performance of the architectural optimization because the higher is the E_c, the faster is an instruction executed.

Figure 8.15 depicts E_c measured for schedule A, B, and C. The computation efficiency degrades as the number of workers increases. This is because the rate

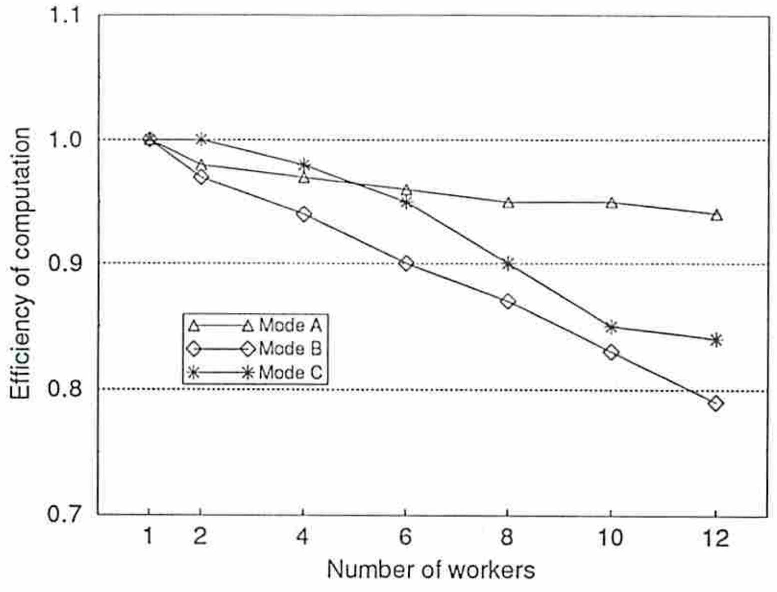


Figure 8.15: Average efficiency of computation of benchmark set I: schedule A, B, C

of remote accesses becomes higher. Schedule A has the highest computation efficiency among the three. The decreasing rate is smaller than those of the other two. In the setting of 12 workers, the values E_c measured for schedule A, B, and C are 0.94, 0.84, and 0.79, respectively. The result indicates that the architectural optimization works effectively in reducing the amount of remote accesses in program execution.

Granularity

A *task* is defined as a continuous work performed by a worker. The execution of a task is carried out by the depth-first search as in the sequential execution. The *granularity* of a task refers to the number of choice points created by a task. The number of task switching and its granularity are listed in Table 8.8, 8.9, and 8.10 respectively for schedule A, B, and C. Overall, the average granularity of a task varies from 5 to 154 for the entire benchmarks. The detailed discussion about the granularity of each benchmark will be made in section 8.3.3.

The average granularity of schedule A, B, and C is depicted in Figure 8.16. The task granularity decreases as the number of workers becomes larger. For schedule B and C, the decrease rate of task granularity becomes lower as the size

Prolog Program	Number of workers									
	4		6		8		10		12	
	T	G	T	G	T	G	T	G	T	G
queens1_8	102	580	147	402	236	250	290	204	498	118
queens2_8	391	319	314	397	402	310	712	175	841	148
tina	158	382	240	252	231	261	395	153	560	108
sm	58	269	97	161	97	161	214	73	242	64
parse2_20	254	120	389	78	557	55	645	47	643	47
parse4_5	355	129	475	96	991	46	855	53	946	48
parse5	1183	228	2593	104	3673	73	3988	67	4908	55
db4_10	95	147	204	68	270	51	348	40	379	36
db5_10	93	184	172	99	317	54	277	61	543	31
house_20	206	242	235	212	310	160	427	116	575	86
parse1_20	147	49	222	32	338	21	499	14	489	14
parse3_20	226	34	278	28	339	23	450	17	474	16
farmer_100	899	29	1270	21	1558	17	1688	15	1860	14
average	320	208	510	150	716	114	829	79	996	60

Table 8.8: Number of task switching and task granularity of benchmark set I: Schedule A

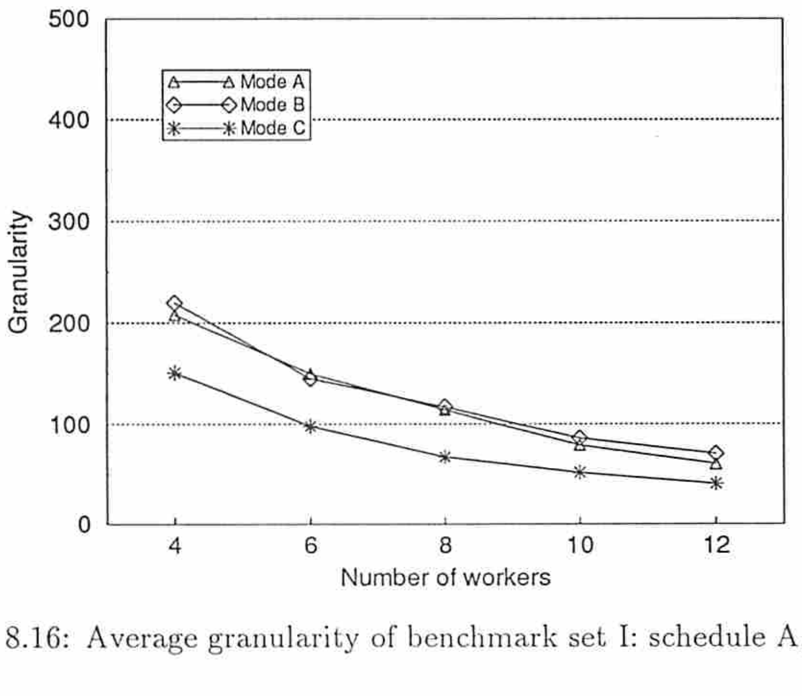


Figure 8.16: Average granularity of benchmark set I: schedule A, B, and C

Prolog Program	Number of workers									
	4		6		8		10		12	
	T	G	T	G	T	G	T	G	T	G
queens1_8	141	419	178	332	197	300	295	200	382	154
queens2_8	259	482	321	389	296	422	532	234	835	149
tina	119	508	248	243	350	172	331	182	477	126
sm	41	381	98	159	121	129	163	96	197	79
parse2_20	472	64	571	53	862	35	952	32	1158	26
parse4_5	318	144	424	108	816	56	929	49	844	54
parse5	1221	221	1529	177	2192	123	2692	100	2518	107
db4_10	94	149	160	87	188	74	322	43	279	50
db5_10	118	145	145	118	251	68	352	48	393	43
house_20	180	277	292	170	471	105	484	103	546	91
parse1_20	312	23	387	18	490	14	541	13	658	11
parse3_20	295	26	472	16	498	15	568	13	721	10
farmer_100	982	27	1574	17	1924	13	2287	11	2632	10
average	350	220	492	145	665	117	803	86	895	70

Table 8.9: Number of task switching and task granularity of benchmark set I: Schedule B

Prolog Program	Number of workers									
	4		6		8		10		12	
	T	G	T	G	T	G	T	G	T	G
queens1_8	123	481	200	295	282	209	406	145	619	95
queens2_8	379	329	504	247	770	162	798	156	1255	99
tina	244	247	366	165	661	91	776	77	1038	58
sm	77	203	147	106	143	109	352	44	237	66
parse2_20	757	40	1130	27	1387	22	1574	19	2152	14
parse4_5	632	72	797	57	1327	34	1351	34	1508	30
parse5	1016	266	1681	161	2926	92	2866	94	3587	75
db4_10	179	78	230	60	275	50	580	24	544	25
db5_10	202	84	277	61	443	38	596	28	655	26
house_20	436	114	672	74	1132	44	1783	27	2127	23
parse1_20	393	18	679	10	799	9	1069	6	1283	5
parse3_20	473	16	876	8	1012	7	1297	6	1471	5
farmer_100	1738	15	2133	12	2941	9	2875	9	3061	8
average	511	151	745	98	1084	67	1255	51	1502	40

Table 8.10: Number of task switching and task granularity of benchmark set I: Schedule C

of the system becomes larger. It is because the effect which a scheduling policy has on the size of the granularity decrease as the number of workers increases.

As pointed out earlier, the granularity of schedule C is the smaller than that of the other two schedules. Although the average task granularity of schedule A and B are almost the same, the granularity of schedule A is smaller than that of schedule B. It is because the architectural optimization affects the top-most scheduling such that sometimes the younger nodes are chosen in task scheduling, which otherwise are not chosen in the top-most scheduling. The rate of difference between average granularity of schedule A and B increases slightly as the number of workers becomes larger. It reflects that the influence of architectural optimization on the top-most scheduling increases as the number of workers becomes larger.

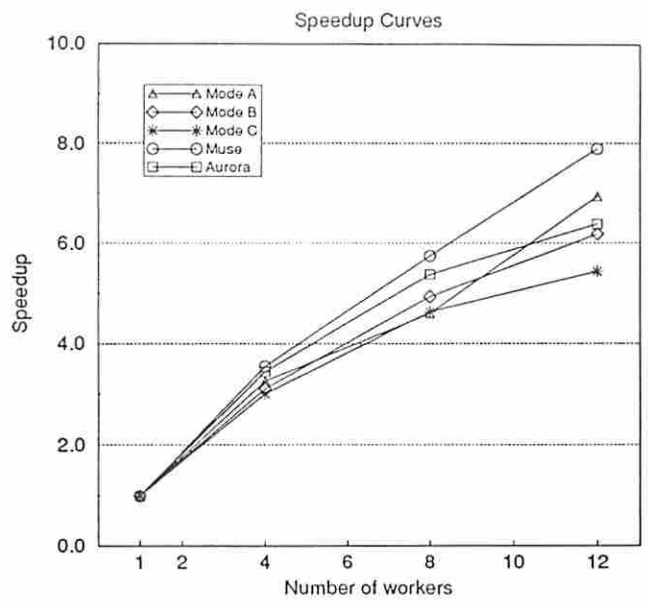


Figure 8.17: Speedup curves of benchmark set I: schedule A, B, C, Muse and Aurora.

Comparison of the performance with other implementation

This subsection presents the performance comparison of our implementation with other prominent implementations. The performance of the Muse and the Aurora is reported in papers [5, 15]. Now that benchmarks used in Muse and Aurora are exactly the same with benchmark set I, a reasonable comparison can be made between their performance and ours.

Figure 8.17 shows the speedup curves for five cases. According to the figure, the Muse shows the best performance. Overall, the performance of the Aurora is better than that of our system. However, our system executing schedule A shows better performance over the Aurora when the number of workers is larger than 10.

8.3.3 Analysis of performance of benchmark set I

In the previous subsection, the average performance of benchmark set I has been analyzed. This subsection presents the performance of each benchmark in benchmark set I and its analysis. The primary objective of the analysis is to identify the relation between the parallelism of benchmarks and the performance of the

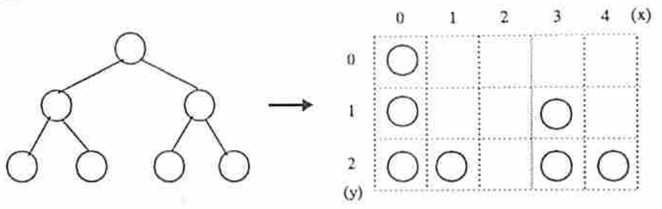


Figure 8.18: Search tree and its signature

system. The discussion will be limited to the case of schedule A, since schedule A generally provides the best performance.

Qualitative analysis of parallelism: signature of the search tree

In an effort to identify the amount of parallelism and to understand execution behavior of a benchmark, we developed a visualization technique which displays runtime search trees. The technique is implemented in our experimental prototype and is used to visualize the search trees of the benchmarks.

In the visualization technique, the search tree is represented by an image called the *signature* of a search tree. For almost all the benchmarks, a search tree consists of an order of ten thousands nodes. It is thus very difficult to represent a search tree in a visual image through which the user can understand the overall shape of the search tree. The signature provides a very accurate visualization of the search tree, while it is very simple and direct. Figure 8.18 shows an example explaining how the signature is represented. In the signature, the nodes of a search tree are represented through a two dimensional grid with two axes X and Y, in which each cell can be mapped to a node. The root of the search tree is mapped onto the upper left corner of the grid. Its child nodes are mapped onto the following row. The X coordinate of a child node is determined by the size of the subtree of the previous child node such that the subtrees of two adjacent child nodes are not overlapped between them. In the example, the two child nodes of the root are mapped respectively onto (1,1) and (1,3). The size of a signature is defined as (x,y) in which "x" and "y" refer respectively to the maximum X and Y values among the coordinates which are occupied by nodes.

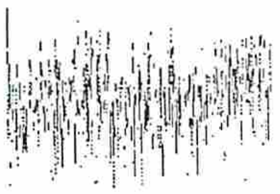
In the signature representation, the level of the search tree is preserved. Also the size of subtree which reflects the parallel grain is clearly depicted because the maximum values of X and Y coordinates represent the size of the search tree and the density of the grid cells onto which nodes are mapped is depicted in the image.

For the implementation of the signature, the trace containing the search tree information is generated in each worker. After the execution of a benchmark, the trace is manipulated to create the search tree and then they are converted into a bit map data corresponding to the signature. The signature is then converted into an image in the PBM image format which can be displayed or converted into a postscript file through general display software such as "xv" in X window systems.

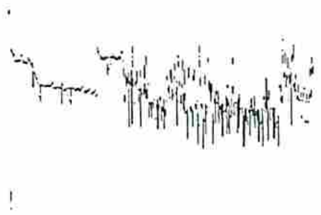
Figure 8.19 and Figure 8.20 present the signatures of all the benchmarks in benchmark set I. The signature exhibits parallelism whose amount is reflected from the size, shape, and density of a signature. The size of the signature for `queen1_8.pl` is (13600,94) and the density in the grid is very high. The size of the signature for `parse2_20.pl` is (24102,99) and the density in the grid is not so high as that of `queens1_8.pl`. Compared with the signature of `queens1_8.pl`, the signature of `parse2_20.pl` shows less parallelism due to lower density. Indeed, the experiment result shows that the average grain size of `queens1_8.pl` is about two times larger than that of `parse2_20.pl`. The signature of `farmer_100.pl` is different from the previous two. The parallelism is not significant because the shape of its signature is vertically long and the number of nodes in each level is absolutely smaller than the previous two. Indeed, the experiment shows that `farmer_100.pl` has the smaller grain size than the previous two.

Speedup

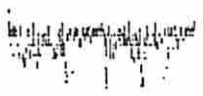
Figure 8.21 shows the speedup curves of the benchmarks in set I. The benchmarks can be crudely classified into three groups in terms of the size of the speedup. The first group includes five benchmarks (`queens1`, `queens2`, `tina`, `parse5`, and `sm`) which show high speedup ranging from 8 to 10. The second group includes six benchmarks (`parse2`, `parse4`, `db4`, `db5`, and `house`) which show medium speedup



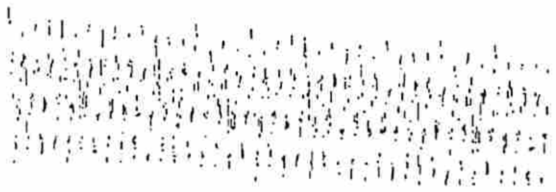
(a)queens1.pl



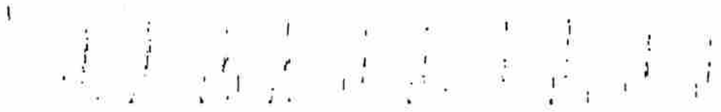
(b) tina.pl: X=21361, Y= 154



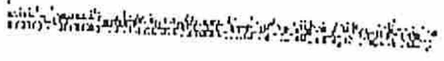
(c) sm.pl: X=6594, Y= 45



(d) parse2_20.pl: X=24102, Y=99

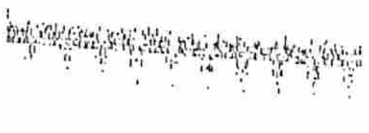


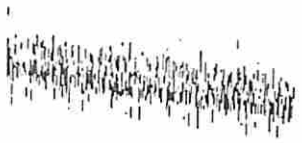
(e) parse4_5.pl: X=36187, Y= 211

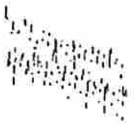


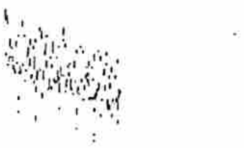
(f) db4_10.pl: X=11659, Y= 29

Figure 8.19: Signatures of benchmark set I (a)


(a) db5_10.pl: X=13859, Y= 44


(b) house_20.pl: X=13207, Y= 72


(c) parse1_20.pl: X=5772, Y= 58


(d) parse3_20.pl: X=6002, Y= 71

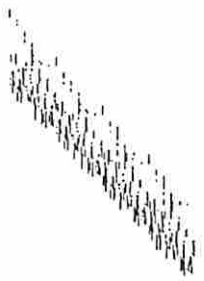

(e) farmer_100.pl: X=10201, Y= 142

Figure 8.20: Signatures of benchmark set I (b)

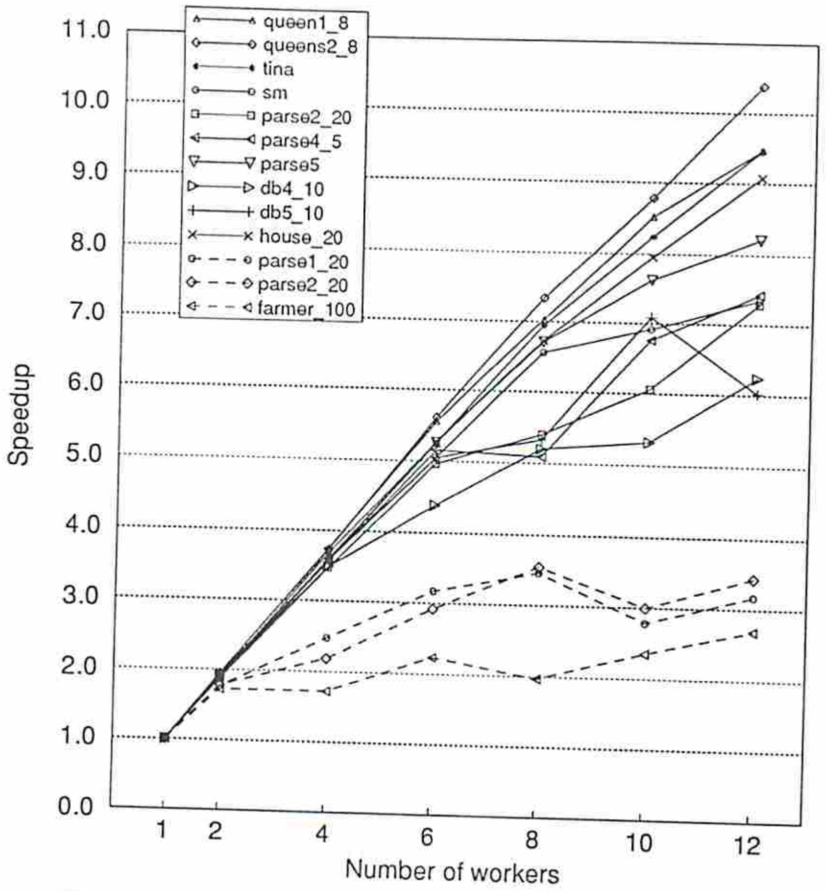


Figure 8.21: Speedup of benchmark set I: Schedule A

ranging from 6 to 7.5. The third group includes the rest of the benchmarks (`parse1`, `parse 2`, and `farmer`) which shows low speedup ranging from 2 to 3.5. The main reason for low speedup of the third group is identified by the lack of parallelism, which will be discussed in more detail shortly.

The SPP IAX system consists of a set of hypernodes connected via SCI rings. The memory access of inter hypernodes is longer than that of intra hypernodes. In the system, when the number of workers becomes larger than eight, the latency of memory accesses becomes longer. Indeed, for most speedup curves, it is noticed that the increase rate of speedups slightly decreases when the number of workers is over eight. It is believed that the decrease reflects the memory organization of the SPP IAX system.

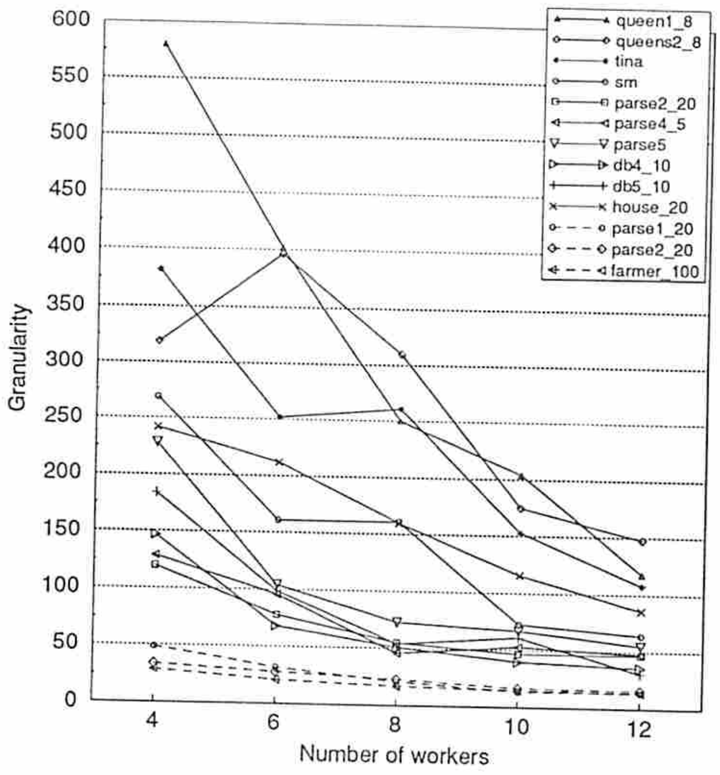


Figure 8.22: Granularity of benchmark set I: Schedule A

Granularity

In the previous subsection, the parallelism of benchmarks was observed by means of a specific representation scheme, *signatures*. Figure 8.22 depicts the curves for the grain size of each benchmark in the seven configuration. The grain size decreases as the number of workers becomes larger. The decrease rate differs from one another. In general, the decrease rate of benchmarks with larger granularity is larger than that of benchmarks with smaller granularity. For the third group of benchmark, the decrease rate of the granularity is hardly noticeable. It is because the amount of parallelism is not sufficient enough to maintain high system utilization when the size of the system is relatively large.

It is also interesting to compare the task granularity between two versions of 8-queens programs, *i.e.,* queens1_8.pl and queens2_8.pl. As for the signatures of the two program, queens1_8.pl has a quite smaller signature than

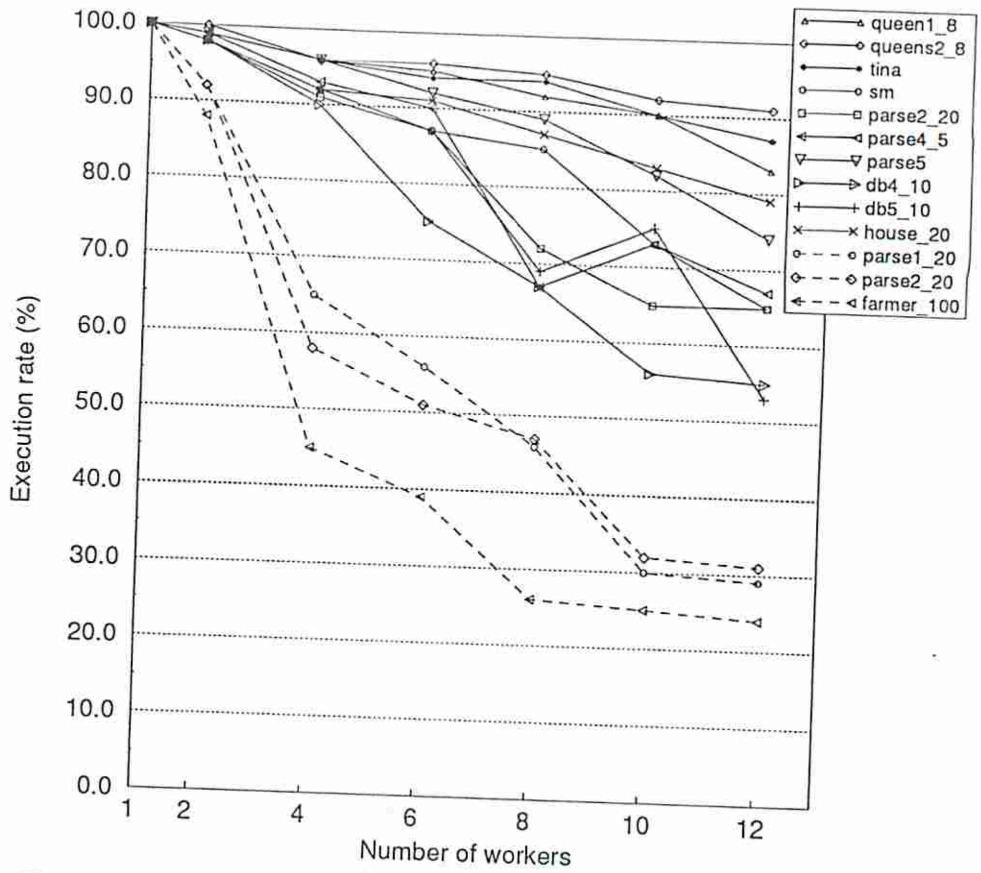


Figure 8.23: Prolog execution rate (%) of benchmark set I: Schedule A

queens1_8.pl. The decrease rate of the granularity of queens1_8.pl is larger than that of queens1_8.pl, because the size reflects the amount of parallelism.

Utilization

Figure 8.23 shows the rate of prolog time over the execution time of each benchmark. The values ranges from abut 25 percents to over 90 percents. They differ from one another depending on the characteristics of benchmarks. The prolog rate is closely related the available parallelism as well as the task switching rate which will be discussed shortly. Particularly, it is shown that the prolog rates of benchmarks in the third group decrease rapidly.

Figure 8.23 shows the task switching rate of each benchmark in set I. In the setting of 12 workers, task switching rates ranges from 5 percents to 18 percents. In general, the task switching rate continues to increase for benchmarks in the

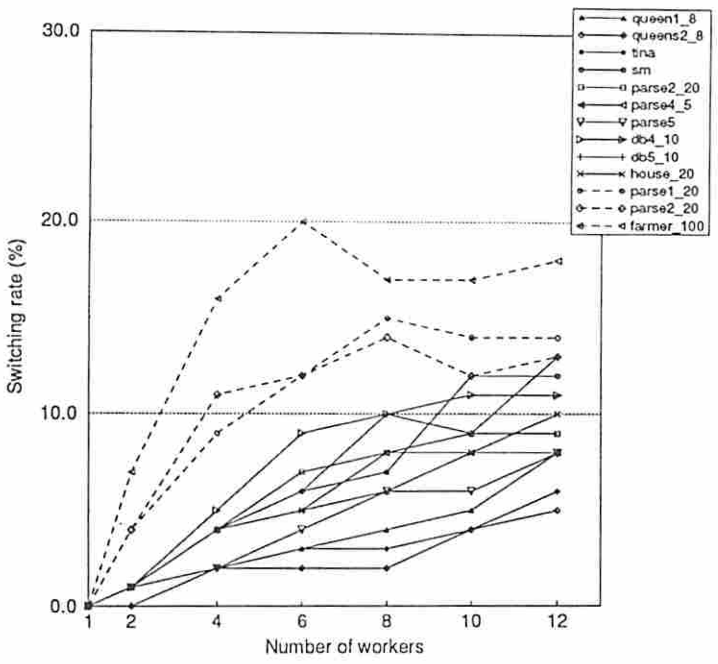


Figure 8.24: Task switching rate (%) of benchmark set I: Schedule A

group one and two, while it stop increasing for benchmarks in the third group when the number of workers reaches some number. The stopping of the increase is because of the lack of parallelism; in the situation of insufficient parallelism, even though the number of workers becomes larger, the total number of task switching remains almost the same since the number of task switching remains the same.

Figure 8.25 shows the idle rate of each benchmark. In the setting of 12 workers, the idle rate of benchmarks in the first group reaches about 60 percents of the execution time. As for the benchmarks in the other two groups, the idle rates are relatively smaller than those for the first group.

8.3.4 Analysis of performance of benchmarks set II

This subsection presents the performance of benchmark set II. As opposed to benchmark set I, benchmark set II contains speculative work to some degree. The objectives of the performance evaluation with benchmark set II are to verify the functionality of our implementation in the presence of speculative work, to

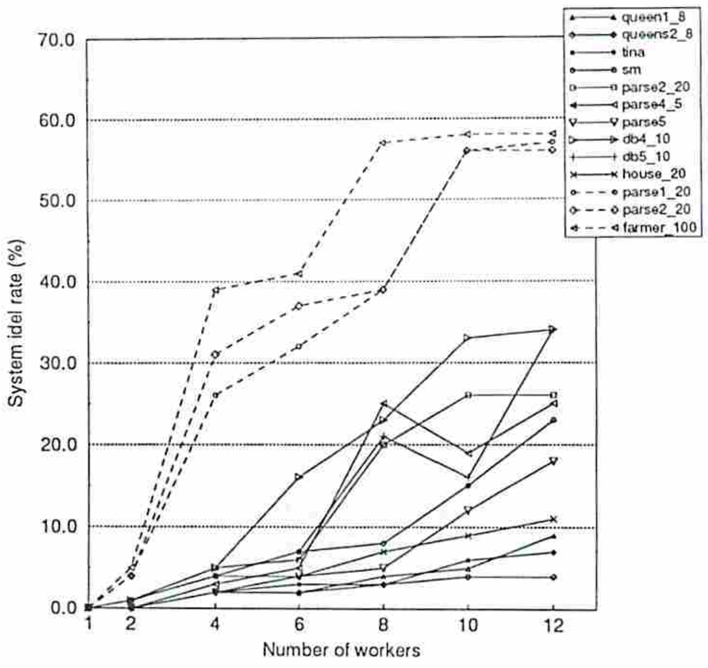


Figure 8.25: Idle rate (%) of benchmark set I: Schedule A

evaluate the performance, and to suggests directions to accomplish the higher speedup.

The speculative work is defined as the work which will be executed in the OR-parallel execution, whereas it will not be executed in the sequential execution. The speculative works are in general produced from the following two sources.

- Single solution problem: The OR-parallel execution is inherently based on a breadth first search. Therefore, it may include the portion of the search tree which is right to the path of the first solution. Because the portion will not be executed in the sequential execution, the work corresponding to the portion becomes speculative work.
- Pruning operation: Even though the portion of the search tree executed by a worker is left to the solution path, it may not be executed during the sequential execution, providing it is pruned out by some “cut” or “commit”. In this case, the work corresponding to the portion becomes speculative work.

The benchmark set II has three programs queens.pl, chat_parser.pl, and zebra.pl. Queens.pl is a single-solution version of the 8-queens problem implementing an algorithm which differs from those in benchmark set I. The program does not have "cut" or "commit" and the speculative work with the program results from the breath-first search. (The all-solution version of the program has 92 solutions.)

Chat_parser.pl is a program made by extracting the part of sentence parsing from the natural language system developed by Fernando C. N. Pereira and David H. D. Warren. The program parses the following list of sentences.

```
What rivers are there ?
Does afghanistan border china ?
What is the capital of upper_volta ?
Where is the largest country ?
Which country's capital is london ?
Which countries are european ?
How large is the smallest american country ?
What is the ocean that borders african countries
and that borders asian countries ?
What are the capitals of the countries bordering the baltic ?
Which countries are bordered by two seas ?
How many countries does the danube flow through ?
What is the total area of countries south of the equator
and not in australasia ?
What is the average area of the countries in each continent ?
Is there more than one country in each continent ?
Is there some ocean that does not border any country ?
What are the countries from which a river flows
into the black_sea ?
```

The speculative work of chat_parser.pl results from the pruning operation by the "cut" in the following clause.

```
determinate_say(X,Y) :-
   say(X,Y), !.
```

In the sequential execution, once a sentence is parsed, it is accepted. The other instances of syntactic parsing are not tried. As a matter of fact, this is implemented through the "cut". In the parallel execution, before the "cut" is executed, other workers may take some branches which may result in the other instances of parsing. The work of this case clearly amounts to speculative work.

`Zebra.pl` is a puzzle solution written by Claude Sammut. It solves the puzzle of "Where does the zebra live?". It contains one "cut" which results in most of speculative work of the program.

Table 8.11 shows the speedups of the above three benchmarks obtained by schedule A, B, and C. The corresponding speedup curves are depicted in Figure 8.26.

In Figure 8.26, the upper three curves are for `chat_parser.pl`, obtained for schedule A, B, and C. The next three curves are for `zebra.pl`.. The lower three curves are for `queens8.pl`.

Different from the speedup curves for benchmark set I, the curves for benchmark set II show highly irregular shapes. Moreover, the scheduling policy does not have any persistent and significant influence on the speedup. For example, in the system with some number of workers, some scheduling policy results in the highest speedup, whereas in the system with another number of workers, a different scheduling policy results in the highest speedup.

Table 8.12 shows the total number of nodes executed for each configuration and its speculative rate. Given a system configuration, the *speculative rate* refers to the rate of the number of the nodes executed in the configuration minus the number of nodes to be executed in the single worker configuration over the number of node executed in the single worker configuration.

Figure 8.27 depicts speculative rates in curves. Compared with the other two programs, the speculative rates of `chat_parser.pl` are very low. In the configuration of 12 workers, they are respectively 15, 6, 9 percents respectively for schedule A, B, and C. The speculative rates of `zebra.pl` are 79, 63, and 77 percents. They are 342, 202, and 346 percents for `queens8.pl`.

Scheduling type	Prolog Program	Number of workers						
		1	2	4	6	8	10	12
Schedule A	queens8	1.00	0.94	1.13	1.73	2.32	1.70	2.43
	zebra	1.00	1.64	2.35	2.69	4.54	5.08	5.78
	chat_parser	1.00	1.82	3.52	4.95	5.00	5.22	5.72
	average	1.00	1.47	2.33	3.12	3.95	4.00	4.64
Schedule B	queens8	1.00	0.94	1.09	1.50	2.95	0.98	2.89
	zebra	1.00	1.64	2.99	3.08	3.37	3.99	5.44
	chat_parser	1.00	1.85	3.41	4.49	5.86	5.59	6.35
	average	1.00	1.48	2.50	3.02	4.06	3.52	4.89
Schedule C	queens8	1.00	1.00	1.13	1.49	1.65	1.99	2.06
	zebra	1.00	1.74	2.25	3.49	4.39	4.19	4.51
	chat_parser	1.00	1.92	3.21	4.63	5.70	6.11	5.91
	average	1.00	1.55	2.20	3.20	3.91	4.10	4.16

Table 8.11: Speedup: benchmark set II

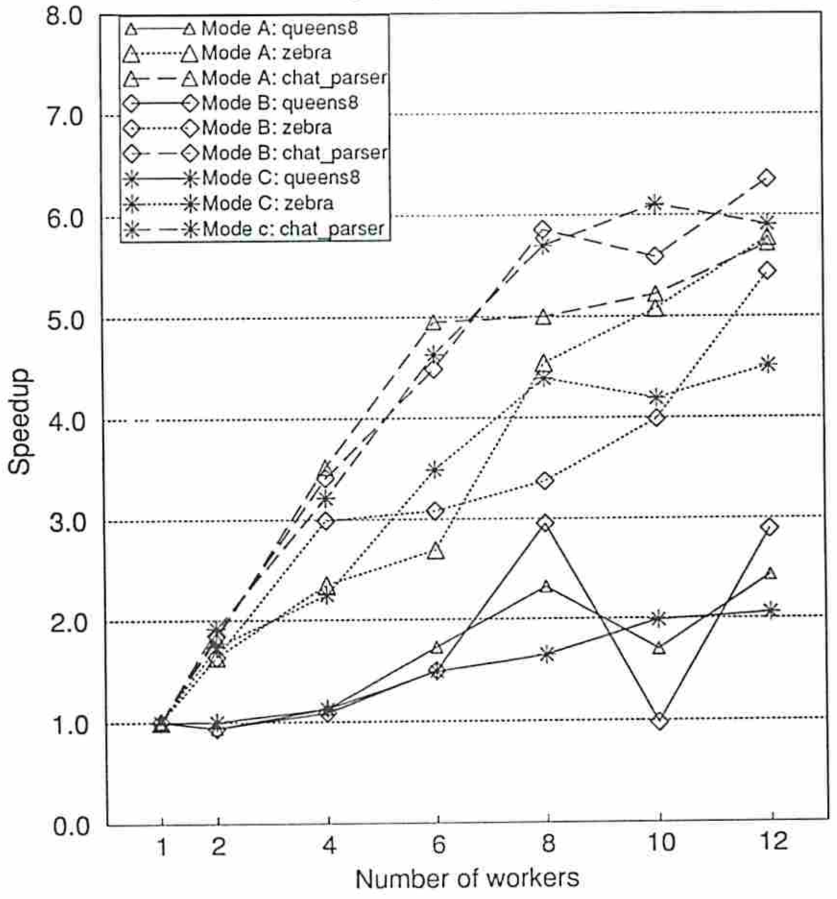


Figure 8.26: The speedup of benchmark set II

Sch. type	Prolog Program	Number of workers				
		1	2	4	8	12
A	queens8	28182 (0)	28516 (1)	94092 (234)	89544 (218)	124687(342)
	zebra	14481 (0)	17430 (20)	23517 (62)	22971 (59)	25935 (79)
	chat.	32603 (0)	34287 (5)	34048 (4)	35728 (10)	37544 (15)
B	queens8	28182 (0)	28516 (1)	95256 (238)	62769 (123)	84980 (202)
	zebra	14481 (0)	17430 (20)	18315 (26)	27412 (89)	23607 (63)
	chat.	32603 (0)	33649 (3)	33930 (4)	33405 (2)	34398 (6)
C	queens8	28182 (0)	28510 (1)	96646 (243)	117298(316)	125712(346)
	zebra	14481 (0)	16434 (13)	24548 (70)	20808 (44)	25681 (77)
	chat.	32603 (0)	33682 (3)	38086 (17)	35405 (9)	35512 (9)

Table 8.12: Total number of nodes and speculative rate (%)

The changes of the speculative rate is not regular, which results in the irregular speedup. As did for speedups, the scheduling policies do not have significant influence on the amount of speculative work.

The above observation indicates that special measures must be taken for high performance in the presence of speculative work. As one of them, a novel scheduling algorithm is crucial which can handle the speculative work in efficient ways. Such algorithms will help to estimate the probability of some tasks to be speculative and to provide some means to select some work which is less probable to be speculative.

8.4 Synthesis

The chapter presents the implementation and the result of performance evaluation. Based on the evaluated performance, the potential of the execution model has been analyzed. The analysis result shows that the execution model is flexible enough to accommodate a wide range of scheduling algorithm as well as is efficient in implementing such scheduling algorithms.

In our experiment, we implemented three algorithms, a pure top-most scheduling, a top-most scheduling with an architectural optimization, and a bottom-most scheduling. The top-most scheduling is advantageous over the bottom-most

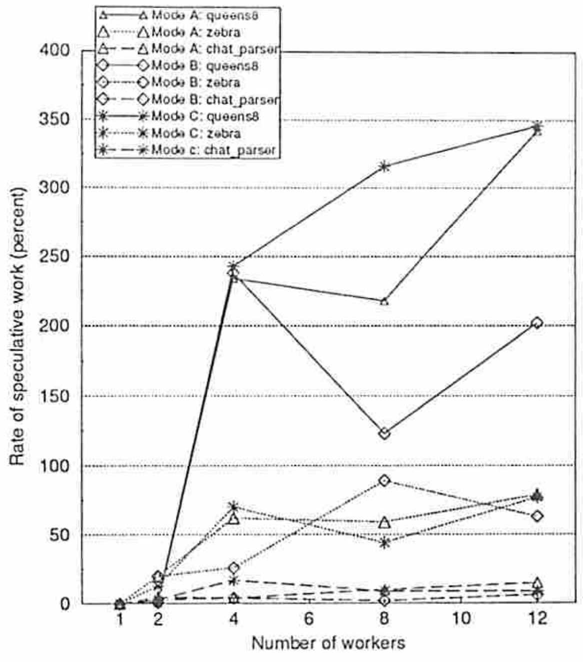


Figure 8.27: The graph of speculative rates

scheduling due to larger granularity for all solution problems with little or no speculative work. However, because of the overhead caused from such operations as finding the top-most nodes, it is difficult to obtain substantial performance gain from the top-most scheduling. In our system, the top-most scheduling produces substantial performance gain over the bottom-most scheduling, which gives us a clue that our execution model is very efficient in supporting a wide range of algorithmic optimizations. Moreover, the architectural optimization to reduce the remote memory accesses shows significant performance gain, even though such optimization may introduce severe overhead which may even cause the risk of performance degradation. This indicates that our execution model efficiently supports such optimizations. The performance evaluation and its analysis validate that our execution model is efficient and flexible for both algorithmic and architectural optimizations.

Chapter 9

Conclusions and future research issues

This chapter concludes the dissertation and offers future research issues.

9.1 Conclusions

In this dissertation, we investigated and addressed a variety of issues pertaining to the design and implementation of parallel logic programming systems on large-scale parallel machines.

Within the context of large-scale parallel logic programming, we noticed that the scheduling issue becomes more important. Normally, the activities performed by a scheduler are tightly coupled with the components provided in the parallel execution models. For example, the cost of task switching depends on the binding environment of the parallel execution model. For high performance in such systems, schedulers must be capable of optimizing architectural features as well as scheduling strategies. Such optimizations include selection of scheduling strategies specific to the characteristics of application programs and also enhancement of locality by reducing the amount of remote accesses. In this dissertation, we have argued that for high performance logic programming on large scale multiprocessors, the issues of flexibility and efficiency associated with scheduling activities must be clearly addressed in the design of the parallel execution model.

In order to validate the argument, we have designed and implemented a parallel execution model on a HP's SPP IAX distributed shared multiprocessor. In an effort to show that a wide range of scheduling algorithms can be implemented with minimal overhead on the execution model, both a top-most scheduling and a bottom-most scheduling have been implemented. Moreover, an architectural optimization particularly to reduce the amount of remote memory accesses has been implemented within a top-most tree-based scheduling strategy. The performance result and its analysis have shown that our execution model provides high flexibility and efficiency both for algorithmic and architectural optimization in the runtime scheduling. Moreover, the result provides a validation that the architectural and algorithm optimization are crucial for expected performance on large-scale parallel machines.

9.2 Future Research Issues

This section presents the directions of future parallel logic programming systems and research issues.

9.2.1 What needs to be achieved

The general directions for future parallel logic programming system are summarized as follows:

1. *The usage of a wider range of parallelism:* In order to get an appreciable speedup on large-scale parallel machines, parallel logic programming systems need to integrate other types of parallelism such as AND-parallelism in addition to OR-parallelism.

2. *High performance scheduler:* The scheduling problem needs to be investigated in the presence of different types of parallelism, particularly AND- and OR-parallelism.

3. *Implementation study:* Beyond simulation studies, the implementation study based on a robust parallel implementation of the OR/AND model on practical large-scale parallel machines is essential for a thorough investigation of the parallel execution behavior and the system performance.

4. *Inter-operability:* In spite of almost a decade of research efforts, commercial parallel logic programming implementations are not yet available. On the parallel software technology side, it is partly because developing parallel software is extremely expensive. The software to be developed in the parallel logic programming implementation must therefore provide enhanced portability and inter-operability. For this, research efforts are needed to establish a framework that makes some software components, *e.g.,* schedulers, readily available to other systems which require the same functionality.

Under the above directions, future research issues are discussed in the following sections.

9.2.2 Integration of independent AND parallelism

The parallel execution model developed in this dissertation deals only with OR-parallelism. As a future research, it is suggested to integrate independent AND-parallelism on top of the current implementation. The issues associated with the integration are outlined as follows:

Investigation of AND/OR parallel model

The addition of AND-parallelism to the OR-parallel execution model brings out several new issues associated with the maintenance of consistent bindings, interaction between AND- and OR-parallelism, and scheduling AND- and OR-tasks, etc. Therefore, it is necessary to investigate a parallel execution model which will address such issues and provide a specification of an parallel abstract machine.

Static analysis of AND-parallelism

For a more efficient implementation of independent AND-parallelism of parallel logic programming systems, it is necessary to detect as much AND-parallelism at compile-time as possible. Static data-dependency analysis, another approach which is most frequently used for the detection of AND-parallelism, is not sufficient to fully detect the AND-parallelism because it has a limitation in dealing with the propagation of groundness. Abstract interpretation provides a great opportunity for more complete detection of AND-parallelism. However, the full-blown abstract interpretation is not efficient because it inherently employs operations which are very complex and not necessary for gathering the information of AND-parallelism. In other words, it generally require very large computation which can be hardly afforded just for the detection of AND-parallelism. For gathering only the information that is essential for detection of AND-parallelism, the approach will be more effective which combines the simplicity of static data-dependency analysis and the completeness of abstract interpretation. To provide a precise and efficient way of the abstract interpretation, it is necessary to investigate an efficient abstract domain and its primitive operations which can be applied universally to PROLOG programs for the detection of AND-parallelism.

9.2.3 Advanced scheduling scheme

Search trees created at runtime for PROLOG programs can be classified into several types. Depending on the characteristics of each type, there exists a specific scheduling strategy which will outperform the other scheduling strategies. To get high performance, we need to identify the characteristics of search trees as well as the interaction between the characteristics and scheduling strategies.

Static analysis of scheduling information

Previous schedulers perform the allocation of tasks only with the information extracted at runtime [14, 15, 17]. Although these schedulers have proven very effective regardless of speculative or non-speculative tasks, higher performance can be achieved by utilizing some information extracted at compile time. For example, if we can predict at compile time the subtrees which will become speculative work and estimate the shapes of search trees, the schedulers can make an efficient selection of scheduling policies. For this, we need to identify the required information and also to investigate methodologies which help analyze the information at compile time.

Speculative and multi-paradigm scheduling

For a given shape of the search tree, a particular scheduling policy may produce the higher performance. For example, the top-down scheduling policy, which would be inferior to the bottom-up scheduling policy for a number of general benchmark programs, would produce better performance for the all-solution problem without "cut" or "commit" predicates because of larger task granularity. For high performance logic programming system, particularly on large-scale parallel machine, multiparadigm scheduling is essential. Mutiparadigm scheduling refers to a scheduling paradigm which uses both runtime and compile time information as well as employs more than one scheduling strategies. In other word, multi-paradigm scheduling exploits both static and dynamic information for task allocation. Moreover, it is capable of choosing a specific scheduling strategy adaptively at runtime according to the execution behavior or dynamic information.

9.2.4 Performance model

A performance model of a Prolog program which enables us to quantify the ideal and worst case parallel performance for a PROLOG programs is useful. For example, we can use it as a guide to evaluate the quality of a schedule. The ideal performance refers to the performance when no PE computes the speculative work, while the worst case refers to the performance when the entire speculative work is computed. The suggested methodology toward such a performance model is to develop a trace tool which generates the execution trace of a PROLOG program such that the trace will include all the speculative as well as non-speculative work. From the trace, we calculate the amount of speculative as well as non-speculative work, and then we calculate the ideal and the worst case performance.

Appendix A

Performance Data

Benchmark class	Prolog Program	Number of workers						
		1	2	4	6	8	10	12
Benchmark set I	queens1_8	839	425	222	156	118	104	92
	queens2_8	1917	965	501	343	265	228	197
	tina	1481	775	402	283	222	192	168
	sm	197	102	55	42	30	26	25
	parse2_20	576	302	165	112	96	81	71
	parse4_5	847	436	240	164	135	112	101
	parse5	4904	2515	1322	937	738	646	559
	db4_10	321	165	92	69	52	49	52
	db5_10	370	190	108	74	62	53	49
	house_20	535	274	144	101	80	67	58
	parse1_20	140	79	58	53	49	44	41
	parse3_20	148	83	62	43	43	45	41
	farmer_100	321	184	156	144	146	128	133
Benchmark set II	queens8	642	662	562	376	286	402	285
	zebra	181	109	77	60	41	35	34
	chat_parser	3727	1970	1028	765	674	572	524

Table A.1: Execution time: Schedule A

Benchmark class	Prolog Program	Number of workers						
		1	2	4	6	8	10	12
Benchmark set I	queens1_8	831	421	225	161	123	109	95
	queens2_8	1925	969	514	352	273	234	216
	tina	1547	808	432	311	250	219	193
	sm	189	96	52	37	32	27	26
	parse2_20	592	309	170	123	107	111	118
	parse4_5	839	431	237	165	135	125	111
	parse5	4904	2516	1321	946	747	639	566
	db4_10	329	169	93	69	56	48	45
	db5_10	370	190	103	77	63	55	50
	house_20	527	268	148	103	81	69	64
	parse1_20	140	78	70	63	54	57	55
	parse3_20	156	87	66	70	58	52	55
	farmer_100	329	202	196	183	184	173	166
Benchmark set II	queens8	634	653	568	595	208	609	159
	zebra	181	110	77	45	46	40	26
	chat_parser	3711	1961	1061	816	715	577	488

Table A.2: Execution time: Schedule B

Benchmark class	Prolog Program	Number of workers						
		1	2	4	6	8	10	12
Benchmark set I	queens1_8	831	420	218	152	122	106	98
	queens2_8	1884	946	489	344	269	230	202
	tina	1432	725	384	280	223	196	176
	sm	206	104	57	40	35	31	27
	parse2_20	576	302	173	154	174	156	141
	parse4_5	831	420	231	162	136	113	126
	parse5	4887	2459	1283	908	730	620	535
	db4_10	304	157	84	64	51	45	60
	db5_10	370	189	103	77	60	56	73
	house_20	518	267	146	109	86	79	107
	parse1_20	132	71	83	79	72	73	71
	parse3_20	148	86	92	84	81	81	74
	farmer_100	313	176	237	212	196	172	163
Benchmark set II	queens8	625	625	553	567	324	300	237
	zebra	181	104	74	44	43	34	32
	chat_parser	3694	1920	1133	801	616	544	489

Table A.3: Execution time: Schedule C

Reference List

[1] A. Aiba, K. Sakai, Y. Sato, D Hawley, and R. Hasegawa. Constraint Logic Programming Language CAL. In *Proceedings of the 2nd International Conference on Fifth Generation Computer Systems*, 1988.

[2] H. Ait-Kaci and R. Nasr. LOGIN: A Logic Programming Language with Built-in Inheritance. *Journal of Logic Programming*, 3:185–215, 1986.

[3] H. Ait-Kaci and A. Podelski. Toward a Meaning of LIFE. *Journal of Logic Programming*, 16:195–234, 1993.

[4] K. Ali. OR-Parallel Execution of Prolog on BC-Machine. In *Fifth International Conference and Symposium on Logic Programming*, pages 1531–1545. MIT Press, 1988.

[5] K. Ali and R. Karlsson. Full prolog and scheduling or-parallelism in muse. *International Journal of Parallel Programming*, 19:445–475, 1990.

[6] K. Ali and R. Karlsson. The muse approach to or-parallel prolog. *International Journal of Parallel Programming*, 19:129–162, 1990.

[7] K. Ali and R. 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, 1990.

[8] K. Ali and R. Karlsson. Scheduling speculative work in muse and performance results. *International Journal of Parallel Programming*, 21:449–476, 1992.

[9] H. Alshawi and D. Moran. The Delphi Model and Some Preliminary Experiments. In *Fifth International Conference and Symposium on Logic Programming*, pages 1578–1589. MIT Press, 1988.

[10] B. The Performance of Parallel PROLOG Program. *IEEE Transactions on Computers*, 39:1434–1445, 1990.

[11] R. Bahgat. *Non-Deterministic Parallel Logic Programming*. PhD thesis, Dept. of Computing, Imperial College of Science and Technology, Feb. 1991.

[12] L. Bic. A Data-Driven Model For Parallel Interpretation of Logic Programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 517–523, 1984.

[13] P. Bosco, C. Cecchi, C. Moiso, M. Port, and G. Sofi. Parallel PROLOG using Stack Segments on Shared-Memory Multiprocessors. In *1984 Symposium on Logic Programming*, pages 2–11, Feb. 1984.

[14] R. Butler, T. Disz, E. Lusk, R. Olson, R. Overbeek, and R. Stevens. Scheduling OR-Parallelism: An Argonne Perspective. In *Fifth International Conference and Symposium on Logic Programming*, pages 1590–1605. MIT Press, 1988.

[15] A. Calderwood and P. Szeredi. Scheduling OR-parallelism in Aurora - the Manchester Scheduler. In *Proceedings of the sixth International Conference and Symposium on Logic Programming*, pages 419–435, 1989.

[16] J. Chang, A. Despain, and D. DeGroot. And-Parallelism of Logic Programs based on Static Data Dependency Analysis. In *Digest of Papers of COMPCON Spring 1985*, pages 218–225, 1989.

[17] C. Chen. *Scheduling Heuristics and Runtime Data Structures for the Parallel Execution of* PROLOG *Programs.* PhD thesis, Dept. of Computer Science, University of California at Berkeley., 1991.

[18] A. Ciepielewski and S. Andrzej. *A Formal Model for OR-Parallel Execution of Logic Programs.* Information Processing 83. Elsevier-North Holland, 1983.

[19] A. Ciepielewski, S. Haridi, and B. Hausman. OR-Parallel PROLOG on Shared Memory Multiprocessors. *Journal of Logic Programming*, 7:125–147, 1989.

[20] K. Clark and S. Gregory. A Relation Language for Parallel Programming. In *Proceedings of ACM Conference on Functional Programming Languages and Computer Architecture*, pages 171–178, October 1981.

[21] K. Clark and S. Gregory. PARLOG: Parallel Programming in Logic. In *ACM Transactions on Programming Languages and Systems*, volume 8, pages 1–49, 1986.

[22] A. Colmerauer. *Prolog III Reference and Users Manual, Version1.1.* Marseilles, 1990.

[23] J. Conery. *The AND/OR Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, Dept. of Computer Science, University of California at Irvine, June 1983.

[24] J. Conery. Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors. In *Proceedings of the International Symposium on Logic Programming*, pages 159–170. IEEE Computer Society Press, September 1987.

[25] J. Conery. Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors. *International Journal of Parallel Programming*, 17:125–152, April 1989.

[26] J. Conery and D. Kibler. AND parallelism in Logic Programs. In *Proceedings of the International Joint Conference in AI*, 1983.

[27] S. Debray, K. Bosschere, and D Gudeman. *Call Forwarding: A Simple Low-Level Code Optimization Technique.* Kluwer, 1993.

[28] D. DeGroot. Restricted AND-Parallelism. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1984.

[29] C. Diaz and D. Diaz. wamcc: Compiling Prolog to C. In *Proceedings of the Joint International Conference and Symposium on Logic Programming.* MIT Press, December 1995.

[30] D. Diaz and P. Codognet. A Minimal Extension of the WAM for clp(FD). In *Proceedings of the 10th International Conference on Logic Programming*, pages 774–790, 1993.

[31] M. Dincbas, P. van Hentenryck, H. Simonis, and A. Aggoun. The Constraint Logic Programming Language CHIP. In *Proceedings of the 2nd International Conference on Fifth Generation Computer Systems*, 1988.

[32] F. Henderson and T. Conway and Z. Somogyi. Compiling logic programs to C using GNU C as a portable assembler. In *Proceedings of the JICSLP'95 Post conference on Implementation Techniques for Logic Programming Languages.* MIT Press, December 1995.

[33] Virtual Memory Support for Parallel Logic Programming Systems. A. Veron and J. Xu, *et al.* In *Proceedings of Conference on Parallel Architectures and Languages Europe.* Springer Verlag, June 1991.

[34] J-L. Gaudiot and H. Kim. Concurrent Logic Programming Language on Data-Driven Architectures. In *Proceedings of FGCS'92 workshop on Future directions of Parallel Programming and Architecture*, June 1992.

[35] D. Gudeman, K. De Bosschere, and S. Debray. jc: An Efficient and Potable Sequential Implementation of Janus. In *Proceedings of Joint International Conference and Symposium on Logic Programming.* MIT Press, 1992.

[36] G. Gupta and V. Costa. IDIOM: A Model for Integrating Dependent-AND, Independent-AND and OR-parallelism. In *Proceedings of the International Logic Programming Symposium*, pages 152–166. MIT press, 1991.

[37] G. Gupta and V. Costa. A Systematic Approach to exploiting Implicit Parallelism in Prolog. In *Proceedings of 26th Hawaii International Conference on System Science*, 1993.

[38] G. Gupta and B. Jayaraman. Analysis of Or-parallel Execution Models. *ACM Transactions On Programming Languages and Systems*, 15:659–680, Sep. 1993.

[39] G. Gupta and B. Jayaraman. AO-WAM : A WAM Execution for Compiled And-Or Parallelism. *Journal of Logic Programming*, 17:59–89, Oct. 1993.

[40] G. Gupta, E. Pontelli, and V. Costa. Shared Paged Binding Array: A Universal Data-Structure for Parallel Logic Programming. In *Proceedings of NSF/ICOT workshop on Parallel Logic Programming*, 1994.

[41] H. Exploitation of Fine-grain Parallelism in Logic Languages on Massively Parallel Architectures. In *Proceedings of international conference on Parallel Architectures and Compilation Techniques*, August 1994.

[42] S. Haridi. A Logic Programming Language Based on the Andorra Model. *New Generation Computing*, 7:109–125.

[43] R. Hasegawa and M. Amamiya. Parallel Execution of Logic Programs based on Dataflow Concept. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1984.

[44] B. Hausman, A. Ciepielewski, and A. Calderwood. OR-parallel PROLOG Make Efficient on Shared Memory Multiprocessor. In *1987 Symposium on Logic Programming*, pages 69–79. IEEE Computer Society Press, August 1984.

[45] B. Haussman. *Turbo Erlang: Approaching the Speed of C.* Kluwer, 1993.

[46] N. Ito, H. Shimizu, M. Kishi, E. Kuno, and K. Rokusawa. Data-flow Based Execution Mechanisms of Parallel and Concurrent PROLOG. In *New Generation Computing*, volume 3, pages 15–41. OHMSHA,LTD. and Springer-Verlag, 1985.

[47] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The clp(r) languages and system. *ACM Transactions on Programming Languages*, 14:339–395, 1992.

[48] V. Janakiram, D. Agarwal, and *et al.* R. Malhotra. A randomized parallel backtracking algorithm. *IEEE Transactions on Computers*, 37, December 1988.

[49] P. Kacsuk. *Execution Models of* PROLOG *for Parallel Computers.* The MIT Press, 1990.

[50] P. Kacsuk. Execution of PROLOG on Massively Parallel Distributed Systems. Technical report, Center for Parallel Computing, Queen Mary and Westfield College, 1991.

[51] P. Kacsuk and M. Wise, editors. *Implementations of Distributed* PROLOG. Parallel Computing. WILEY, 1992.

[52] L. Kale. The Reduced-OR Process Model for Parallel Execution of Logic Programs. *Journal of Logic Programming*, 11:55–84, 1991.

[53] L. Kale, B. Ramkumar, and W. Shu. A Memory Organization Independent Binding Environment for And and Or Parallel Execution of Logic Programs Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 1223–1240. MIT Press, August 1988.

[54] J. Kergommeaux. An Abstract Machine to Implement OR-AND Parallel PROLOG Efficiently. *Journal of Logic Programming*, 8:249–264, 1990.

[55] H. Kim and J-L. Gaudiot. QCE: A Binding Environment for Parallel Logic Programming for Large-Scale Multiprocessors . Technical Report Number: CENG-94-09, EE-system, University of Southern California, January 1994.

[56] H. Kim and J-L. Gaudiot. Analysis of Static Binding Environment for Large-Scale Parallel Logic Programming. Technical report, EE-system, University of Southern California, August 1995.

[57] R. Kowalski. *Logic for Problem Solving.* Elsevier North-Holland, 1979.

[58] K. Kumon, H. Masuzawa, and A Itashiki. Kabu-Wake: A New Parallel Inference Method and its Evaluation. In *The Twenty-first IEEE Compute Society International Conference (COMPCON' 86)*, pages 168–172, 1986.

[59] Y. Lin and V. Kumar. AND-Parallel Execution of Logic Programs on a Shared-Memory Multiprocessor. *Journal of Logic Programming*, 10:155–178, 1991.

[60] G. Lindstrom. OR-Parallelism on Applicative Architectures. In *Second International Logic Programming Conference*, pages 159–170, 1984.

[61] J. Lloyd. *Foundation of Logic Programming.* Spring-Verlag, 1987.

[62] A. Mantssivoda. Flang and its Implementation. In *Proceedings of the symposium on Programming Language Implementation and logic programming*, pages 151–166. LNCS 714, 1993.

[63] D. Miller. A Logic Programming Language with Lambda-abstraction, Function Variables, and Simple Unification. In *Proceedings of the International Workshop on the Extension of Logic Programming*, pages 253–281. Springer-Verlag LNCS 475, 1991.

[64] D. Miller and G. Nadathur. Higher-Order Logic Programming. In *Proceedings of the 3rd International Conference on Logic Programming*, 1986.

[65] K. Muthukumar and M. Hermenegildo. Complete and Efficient Methods for Supporting Side-effects in Independent/Restricted AND-parallelism. In *Proceedings of the sixth International Conference and Symposium on Logic Programming*, pages 80–97, 1989.

[66] A. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa, and T. Chikayama. Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proceedings of the sixth International Conference and Symposium on Logic Programming*, pages 436–451, 1989.

[67] W. Older and F. Benhamou. Programming in CLP(BNR). In *Proceedings of the Workshop on Principles and Practice of Constraint Programming*, 1993.

[68] F. Pfenning. *Logic Programming in the LF Logical Framework, in Logical Frameworks.* G. Huet and G Plotkin (eds) Cambridge University Press, 1991.

[69] M. Rawling. GHC on the CSIRAC II Dataflow Computer. Technical Report TR-DB-91-05, Division of Information Technology, CSIRO, Australia, July 1991.

[70] T. Reynolds and S. Delgado-Rannauro. *BRAVE - A Parallel Logic Language for Artificial Intelligence*, volume 4. Kluwer Academic, 1989.

[71] V. Saraswat, K Kahn, and J. Levy. Janus: A step towards distributed constraint programming. In *Proceedings of North American Conference on Logic Programming.* MIT Press, 1990.

[72] E. Shapiro. Concurrent PROLOG: A Progress Report. *IEEE Transactions on Computers*, 19:44–58, August 1986.

[73] Z. Somogyi, F. Henderson, and T. Conway. The implementation of mercury, an efficient purely declarative logic programming language. In *Proceedings of the ILPS'94 Post conference Workshop on Implementation Techniques for Logic Programming Languages.*

[74] P. Tinker and G. Lindstrom. A Performance Oriented Design for OR-parallel Logic Programming. In *Proceedings of the fifth International Conference on Logic Programming*, pages 601–615, May 1987.

[75] K. Ueda. Guarded Horn Clauses. In *Logic Programming'85*, volume 221, pages 168–179. Springer-Verlag, October 1986.

[76] P. van Hentenryck, V. Saraswat, and Y. Deville. Design, Implementations and Evaluation of the Constraint Language cc(FD). Technical Report CS-93-02, Brown University, 1993.

[77] P. Voda. The Constraint Language Trilogy: Semantics and Computations. Technical report, Complete Logic Systems, 1988.

[78] D. Warren. An Abstract Prolog Instruction Set. Technical Report Technical Note 309, SRI, October 1983.

[79] D. Warren. Efficient PROLOG Memory Management for Flexible Control Strategies. In *New Generation Computing*, volume 84, pages 361–369, 1984.

[80] D. H. D. Warren. OR-Parallel Execution Models of Prolog. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT'87)*, pages 243–259. Springer-Verlag, March 1987.

[81] D. H. D. Warren. The SRI Model for OR-parallel Execution of PROLOG - Abstract Design and Implementation Issues. In *Proceedings of the International Symposium on Logic Programming*, pages 92–102. IEEE Computer Society Press, September 1987.

[82] H. Westphal, P. Robert, J. Chassin, and J-C. Syre. The PEPSys Model: Combining Backtracking, AND- and OR-parallelism. In *Proceedings of the International Symposium on Logic Programming*, pages 436–448. IEEE Computer Society Press, September 1987.

[83] R. Yand and H. Aiso. P-Prolog: A Parallel Logic Language based on Exclusive Relation. In *Proceedings of the 3rd International Conference on Logic Programming*, pages 255–269, July 1986.