

**Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596**

**ARCHITECTURAL SUPPORT FOR CONCURRENT
LOGIC PROGRAMMING LANGUAGES**

Leon Alkalaj

**August 1989
CSD-890047**

UNIVERSITY OF CALIFORNIA
Los Angeles

Architectural Support for Concurrent Logic Programming Languages

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science


by

Leon Alkalaj

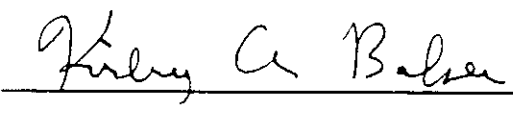
1989

© Copyright by
Leon Alkalaj
1989

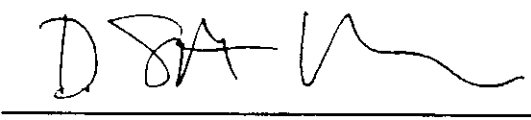
The dissertation of Leon Alkalaj is approved.



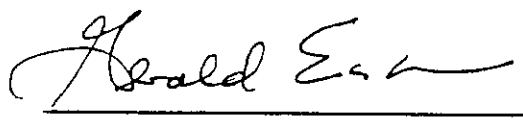
Bruce Rothschild



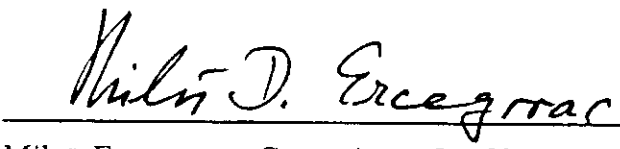
Kirby Baker



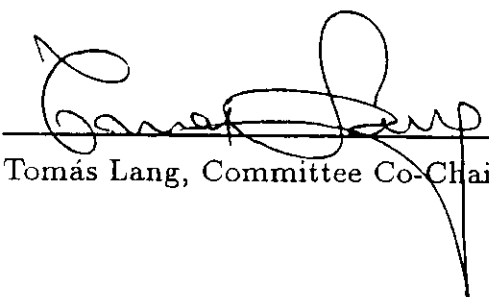
D. Stott Parker



Gerald Estrin



Miloš Ercegovac, Committee Co-Chair



Tomás Lang, Committee Co-Chair

University of California, Los Angeles

1989

To my wife Lea.

To my late aunt, Rene-Lela Kohn.

TABLE OF CONTENTS

1	Introduction	1
1.1	Motivation	1
1.2	Research Objective	1
1.3	Contributions	2
1.4	Outline of Dissertation	5
2	Concurrent Logic Programming Languages	9
2.1	Logic Programming	9
2.2	Parallel AND-OR Execution Models	16
2.3	Sequential AND-OR Execution Model	18
2.4	Parallel Prolog Execution Model	20
2.5	Concurrent Logic Programming	21
2.5.1	Committed-Choice Process Model	25
2.6	Flat Concurrent Logic Programming Languages	26
2.7	An Execution Model of Flat Concurrent Prolog	28
2.7.1	FCP Clause-Try	30
2.7.2	Goal Suspension and Activation	31
2.7.3	Goal Reduction	33
2.8	FCP Programming Examples	34
2.9	Chapter Summary	35
3	Related Work	37

3.1	Special-Purpose Single-Processor Architectures	38
3.2	Parallel Inference Machines	41
3.3	Shared Memory Inference Machines	50
3.4	Chapter Summary	54
4	Implementations of FCP	57
4.1	FCP Sequential Interpreter Implementation	57
4.1.1	FCP Interpreter Characteristics	61
4.2	FCP Sequential Abstract Machine Implementation	61
4.2.1	Abstract Machine Instructions	64
4.2.2	FCP Abstract Machine and WAM	66
4.2.3	Program Representation	67
4.2.4	Performance of Abstract Machine Implementation	70
4.2.5	Compiler Optimizations	71
4.3	A Distributed Implementation of FCP	73
4.3.1	FCP Distributed Interpreter	75
4.3.2	FCP Distributed Abstract Machine	76
4.4	Summary	78
5	Special-Purpose Architectural Support: Design Approach and Results of Analysis	81
5.1	Architectural Support	81
5.2	FCP Processor Design Approach	83
5.2.1	From Operational Semantics to Machine	83
5.3	FCP Implementation Analysis Tradeoffs	84
5.3.1	Proposed Implementation Level Analysis	86
5.4	Empirical Analysis of a Specific System Workload	90

5.4.1	Selected FCP Benchmarks	91
5.4.2	A Workload Session	94
5.4.3	Previously Used Benchmarks	95
5.5	Analytic Performance Evaluation of Hypothesized Bottlenecks . .	96
5.5.1	Analytic Performance Models	96
5.5.2	Redundant Clause Selection	98
5.5.3	Goal Suspension, Activation and Management	102
5.5.4	Dereferencing	115
5.5.5	Clause-Try Trailing	118
5.6	A General Goal Reduction Performance Model	124
5.7	Summary	128
6	FCP Processor Architectural Model	129
6.1	FCP Processor Top-Level Organization	130
6.2	FCP Processor Instruction Set	132
6.3	FCP Processor Interpretive Mechanism	133
6.4	Reduction Unit	136
6.4.1	RU Register Storage	136
6.4.2	RU Instruction Set	139
6.5	Tag Unit	141
6.6	Goal Management Unit	143
6.7	Instruction Unit	146
6.8	Goal Cache	147
6.9	Data Cache	149
6.10	Memory Modules	153
6.10.1	Goal Memory	153

6.11	Chapter Summary	154
7	Overlapped Goal Reduction and Goal Management	155
7.1	Overlapped GMU Execution	155
7.2	RU-GMU Interface	159
7.2.1	Zero Cycle Delay	160
7.2.2	Goal Suspension and Activation: A Global View	162
7.3	GMU Instruction Execution Using the Goal Cache	164
7.3.1	Goal Termination	165
7.3.2	Goal Spawning	167
7.3.3	Goal Suspension	168
7.3.4	Goal Activation	170
7.4	Examples of GMU Execution using the Goal Cache	171
7.5	Properties of GMU Execution using GC	174
8	Analytic Performance Evaluation of RU-GMU Execution	177
8.1	RU-GMU Performance Measures	177
8.2	System Organization and Parameters	178
8.3	Performance Parameter Measurement	181
8.4	Performance Model	183
8.4.1	Average Instruction Execution Time	183
8.4.2	Average RU-GMU Wait Time, \bar{W}	184
8.4.3	Relative Effective GMU Execution Time, R_e	186
8.4.4	GMU and RU Utilizations, U_{gmu} , U_{ru}	187
8.5	Performance Model Parameter Values	189
8.6	Performance Model Analysis	191

8.6.1	Average RU-GMU Wait Time, \bar{W}	193
8.6.2	Relative Effective Execution Time, R_e	198
8.6.3	RU and GMU Utilization, U_{ru}, U_{gmu}	199
8.6.4	Chapter Summary	200
9	FCP Processor Performance Evaluation	203
9.1	Implementation Dependent Parameters	203
9.2	Performance Improvements due to Functional Units	205
9.2.1	Support for Goal Management	207
9.2.2	Support for Data Trailing	207
9.3	Performance versus Goal Management Complexity and Goal Reduction Granularity	209
9.4	Overlapped Goal Management versus Granularity and Complexity	214
9.4.1	Increasing RU Speed of Execution	217
9.4.2	Scaling Granularity	220
9.4.3	Modeling Communication Protocols	223
10	Thesis Summary, Conclusions and Future Work	225
10.1	Future Work: Shared Memory Multiprocessor	227
	References	233

LIST OF FIGURES

2.1	Goal-Head Unification	12
2.2	Don't Care and Don't Know Non-Determinism	13
2.3	AND/OR Computation Tree of a Logic Program	15
2.4	AND-OR Process Model	17
2.5	Prolog Computation Tree: Sequential Search	19
2.6	Prolog Clause-Try	20
2.7	Guarded Horn Clause Computation Tree	22
2.8	Committed-Choice Process Model	25
2.9	Flat Guarded Horn Clause Computation Tree	27
2.10	Program Resolvent: Non-Deterministic Goal Reduction	29
2.11	FCP Clause-Try	31
2.12	FCP Goal Suspension	32
2.13	FCP Goal Reduction	34
3.1	The PIM-D Machine	42
3.2	The PIM-R Machine	43
3.3	The Parallel Inference Machine for Kabu-Wake Execution	45
3.4	PSI-II Machine Organization	47
3.5	Multi-PSI Organization	48
3.6	The Parallel Inference Engine for FLENG	49
3.7	The Aquarius Multiprocessor Machine	54
4.1	FCP Program Interpretation	58

4.2	FCP Interpreter Run-Time Environment	59
4.3	Goal Record Selection	60
4.4	Clause-Tries in Textual Order	61
4.5	Interpreter and Compiler Oriented Machine Implementation	62
4.6	FCP Abstract Machine Run-Time Environment	63
4.7	Goal Suspension Mechanism	67
4.8	FCP Abstract Machine Compiler: Evaluate Interpreter	68
4.9	FCP Procedure Encoding	69
4.10	FCP Clause Types	69
4.11	Compiling FCP Clause Types	71
4.12	Decision Tree Compilation	72
4.13	Remote References in a Distributed Execution Environment	74
5.1	FCP Implementation via Abstract Machine Emulation	85
5.2	FCP Program Analysis Approach	88
5.3	System's Development Workload Session	94
5.4	Analytic Performance Models	97
5.5	Redundant Clause-Try Relative Execution Time	101
5.6	Distribution of the Number of Goal Arguments	107
5.7	Distribution of the Number of Suspension Variables	108
5.8	Distribution of Goals Activated at Clause-Commit	110
5.9	Relative Execution Time $O_{gm}, N_{ct}^r = 5.8$	112
5.10	Relative Execution Time $O_{gm}, N_{ct}^r = 1$	113
5.11	Distribution of Dereference Length	116
5.12	Relative Execution Time of Argument Dereferencing	117
5.13	Distribution of Trail Size at Clause-Commit	121

5.14	Distribution of Trail Size at Clause-Suspend	122
5.15	Trailing at Clause-Failure	122
5.16	Relative Execution Time of Clause Trailing, O_t	123
5.17	Goal Reduction Model	125
6.1	Considerations for the FCP Processor Architectural Model	130
6.2	FCP Processor Multi-Functional Unit Organization	131
6.3	Instruction Execution for Multiple Functional Units	133
6.4	State Diagram of RU-GMU Execution Model	135
6.5	RU,TU and GMU Instruction Execution	135
6.6	RU Addressable Registers During Goal Reduction	138
6.7	TU Tag Setting and Loading	143
6.8	GMU Organization	145
6.9	IU Instruction Prefetching	146
6.10	Goal Cache Organization	147
6.11	Data-Trail Cache Policy: Delayed Binding	151
6.12	Data Trail Policy	152
7.1	Overlapped: a) get/halt; b) put/spawn	156
7.2	Enabling the Overlapped Execution of Suspend	157
7.3	Enabling the Overlapped Execution of Commit	158
7.4	Overlapped Goal Management and Goal Reduction	159
7.5	RU-GMU Interface via Goal Window Pointers	161
7.6	RU Interpretation of GMU Instructions	161
7.7	FCP Processor Execution of Overlapped Goal Suspension	162
7.8	FCP Processor Execution of Overlapped Goal Activation	163

7.9	Goal Termination Algorithm	166
7.10	Goal Termination that Results in GC Underflow	166
7.11	Goal Spawning Algorithm	167
7.12	Goal Spawning that Results in GC Overflow	168
7.13	Goal Suspension Algorithm	169
7.14	Goal Suspension Algorithm	171
7.15	Goal Activation Algorithm	172
7.16	Goal Activation Algorithm	173
7.17	Quicksort Program	173
7.18	Executing the Quicksort Program in the Goal Cache	175
8.1	RU-GMU Performance Model System Organization	179
8.2	Performance Parameter Measurement Approach	182
8.3	RU-GMU Instruction Wait Time w_i	185
8.4	GMU Instruction Distance Distribution	193
8.5	Average Wait Time, \bar{W}	194
8.6	Relative Effective GMU Execution Time, R_e	198
8.7	RU Utilization, U_{ru}	199
8.8	GMU Utilization, U_{gmu}	200
9.1	Relative Execution Times of Goal Reduction Functions	206
9.2	Relative Execution Times with Support for Goal Management	207
9.3	Relative Execution Times with GMU and Data Trail Support	208
9.4	Performance of Different Application Domains	212
9.5	Maximum Speedup for Different Application Domains	215
9.6	The Effect of a Scaled RU on Program Execution Time	218

9.7	Scaled RU Execution Vs. Overlapped GMU	218
9.8	Speedup Versus Increased RU Speed of Execution	219
9.9	Speedup Versus Scaled Goal Management Complexity	222
9.10	Program: Lord of the Rings	223
10.1	Distribution of the Active Goal Queue Size	228
10.2	A Shared Memory FCP Processor Architecture	230
10.3	Load Balancing of Goals	231

ACKNOWLEDGEMENTS

There are many people that I would like to thank for contributing to my Doctoral thesis. It seems only appropriate that I should start, from the *beginning*.

I would first like to thank my parents: mother, Vera (“*ljubi ga majka ...*”) and father, Jozef. Without them, none of this would have been possible. I would like to thank my big-brother, Peter, for being an inspiration and a role-model, and my aunt Beba-Klara Alkalaj for her support.

There are many people that I would like to thank at UCLA. Most of all, my advisors: Prof. Miloš Ercegovac and Tomas Láng. Their wisdom, dedication to work, and honesty will continue to inspire me, for years to come. They have set a standard of advising that I will try to maintain, and to teach others.

The years of work on my PhD would have been more difficult, had it not been for all my friends. I would particularly like to thank Jaime Moreno, Miquel Huguet, T. M. Ravi, Marc Tremblay, Paul Tu, Art Goldberg, Frank Shaffa, Milan Kovačević and Jeong-A Lee, for all the fruitful (and not so fruitful) discussions. A special thank you to Verra Morgan, Dorris Sublette and Saba Hunt.

I would like to thank Prof. Gerald Estrin for taking a special interest in my work and for finding time for long meetings. I thank both Prof. Gerald Estrin and Prof. Eli Gafni for advising me to visit the Weizmann Institute of Science, and work with Dr. Ehud Shapiro.

The visit to the Weizmann Institute and working with Dr. Ehud Shapiro resulted in my dissertation topic. I am immensely grateful to him for being a great *outside-advisor*, a dear friend, and for generously sponsoring my two visits to Israel.

At the Weizmann Institute, I had the pleasure of working with Shmuel Kliger, Avshalom Houri and Bill Silverman. They have significantly contributed to my work with many useful discussions. In particular I thank Avshalom for his patience during the development of the *Statistics Logix* system. Also thanks to Michael Hirsch, Jaakov Levy, Muli Safra, Steve Taylor, Eyal Yardeni, David Weinbaum and John Gallagher. In addition, I would like to thank Lesley, Sara and Dina for their friendship and hospitality.

Finally, this thesis is dedicated to two people. First, to my wife Lea. It is not easy to support a husband-student, and to do it with love and affection. Thank you for your patience, understanding and love. Second, my late aunt, Rene-Lela Kohn. Unfortunately Lela did not live to see her nephew become a Doctor. This was her dream and ambition. Her generosity, love and devotion were unique. Thank you.

VITA

- December 25, 1958. Born, Belgrade, Yugoslavia
- 1977 - 1982 Diploma of Electrical Engineering
Faculty of Electrical Engineering
University of Belgrade
Belgrade, Yugoslavia
- 1983 - 1986 M.S. Computer Science
University of California at Los Angeles
Los Angeles, California
- 1984 - 1987 Student Supplement
IBM Scientific Center
Los Angeles, CA. 90024
- 1983 - 1987 Teaching Associate
UCLA Computer Science Department
Los Angeles, CA. 90024
- 1986 - 1989 PhD. Computer Science
University of California at Los Angeles
Los Angeles, California
- Summer 1987 and 1988 Visiting Scientist
Weizmann Institute of Science
Applied Mathematics Dept.
Rehovot, Israel

PUBLICATIONS and PRESENTATIONS

1. L. Alkalaj, "Flat Concurrent Prolog Abstract Machine Characteristics,"
UCLA Technical Report, CSD-890019.
2. L. Alkalaj, E. Shapiro, "An Architectural Model for A Flat Concurrent
Prolog Processor," *5th Logic Programming Conference/Symposium*, Seattle
1989.

3. L. Alkalaj, A. Bond, "Parallel Logic Programming in CAD/CAM Applications," Manufacturing Engineering Program, Technical Report, October 1988.
4. L. Alkalaj, "A Review of Proposed Architectures for the Execution of Concurrent Logic Programming Languages: An Extended Abstract," *Proceedings of the WesCon/88 Conference*, Anaheim, CA, 1988.
5. L. Alkalaj et. al., "Action Management for a LAN Resource-Sharing System," *6th International Phoenix Conference on Comp. and Comm.*, pp. 469-475.
6. L. Alkalaj et. al., "Action Management for the RM Resource-Sharing System: Providing User's Interface to Services," *IBM LA Scientific Center*, TR G320-2801, 1986.
7. L. Alkalaj et. al. "A Dynamic Memory Management Policy for FP," *20th Hawaii International Conference on System Sciences*, Vol.1., pp. 350-361.
8. L. Alkalaj, "A Uniprocessor Implementation of FP Functional Language," Master's Thesis, UCLA TR CSD-860064, April 1986.
9. L. Alkalaj, "An Analysis Of A Microprocessor Controlled PABX," *Diploma Thesis*, University of Belgrade, Yugoslavia, 1982.

ABSTRACT OF THE DISSERTATION

Architectural Support for Concurrent Logic Programming Languages

by

Leon Alkalaj

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1989

Professor Miloš Ercegovic, Co-Chair

Professor Tomás Lang, Co-Chair

We propose a special-purpose processor and shared-memory multiprocessor architecture for the efficient execution of Flat Concurrent Prolog (FCP). Our design method is based on the analysis of the following suspected implementation bottlenecks: the overhead of *redundant* clause-tries, goal suspension, activation, argument dereferencing, and clause-trailing. The analysis consists of a set of performance models that are part of a general goal reduction model.

We evaluate the models using program parameters obtained empirically by executing the System's Development Workload, which includes programs like the Logix Operating System, FCP Compiler, FCP Processor Simulator, Program Analyzer and Debugger. Measurements are obtained using the *Statistics Logix* (*Slogix*) emulator.

The most significant implementation bottleneck is the overhead of goal management. This includes goal creation, termination, suspension and activation. We characterize the workload as a collection of tightly-coupled fine-grain computations for the cooperative execution of a task.

Based on the analysis, we propose a special-purpose processor and multiprocessor architecture for FCP. The processor consists of multiple concurrent functional units. The main feature is the overlapped execution of goal reduction, in the Reduction Unit, and goal management, in the Goal Management Unit. Goal management operations execute efficiently using a Goal Cache. The overhead of data trailing is reduced using a Data Cache with the *Delayed Binding* policy. The Instruction Unit prefetches instructions for goals stored in the Goal Cache. This enables the execution of goal management operations without losing processor cycles.

In a multiprocessor configuration, Goal Management Unit enables overlapped load balancing and goal sharing. Two simulators of FCP execution (written in FCP) were developed. The first simulates the multi-functional unit architecture, and the second the shared-memory multi-processor architecture.

We evaluate the proposed architecture using analytic performance models. The Goal Management Unit reduces the overhead of goal management from 50% to less than 4% of program execution time, compared to a system that does not have support for goal management. Using the models, we evaluate the

relative execution time of functions, which enables the analysis of implementation bottlenecks.

In the analysis, we also consider other workloads. Overlapped execution of goal management results in higher speedups if the granularity of goal reduction is finer compared to the granularity of goal management. Distributed algorithms and communication protocols exhibit this property. Overlapped execution of goal management, and other features of the FCP processor, are also applicable to Flat Concurrent Logic Programming languages like Flat Parlog or FGHC.

CHAPTER 1

Introduction

1.1 Motivation

A number of high-level programming languages have been proposed with the objective to model the cooperative execution of communicating units of work called *processes* [Hoar85], *actors* [Agha86], *objects* [Gold86] or *threads* [Birr87]. Recently, in the field of logic programming, a group of Concurrent Logic Programming languages have been defined that model the concurrent computation of units of work called *goals*. In particular, we consider the programming language called Flat Concurrent Prolog or FCP [Shap83]. Whereas the different languages represent a variety of programming paradigms, they can be characterized as a collection of tightly-coupled, fine-grain concurrent computations that cooperatively perform a task.

Due to the *mismatch* between the abstract computation models of the previously mentioned programming paradigms, and the execution mechanism of general-purpose machines, their combination results in poor performance. As a consequence, the full power of the programming languages is seldomly achieved. To enable a more efficient execution mechanism, the design of special-purpose environments have been proposed. For example, the Transputer processor is designed for the efficient execution of Occam [Whit85], the J-Machine for CST [Dall88], the Firefly workstation for the Topaz system [Thac88], and so on.

1.2 Research Objective

One way to improve the performance of FCP relative to the sequential implementation on a single general-purpose processor is to define a parallel execution mechanism and then execute FCP on a system of parallel, general-purpose processors. This approach was described in [Tayl89], where FCP is implemented on the Hypercube machine. However, the tradeoff is that the parallel execution model incurs an overhead compared to the sequential execution model running on

a single processor, and second, the complexity of the parallel control mechanism can result in a large overhead of inter-process communication and synchronization. Preliminary performance analysis of some FCP *programming stereotypes* result in performance degradation rather than speedup, when executed on a 16-node Hypercube compared to the sequential execution [Tayl88]. These results motivate research in the following two directions: A more efficient parallel execution mechanism that takes into account the tradeoffs between the complexity of the execution model and the overhead of concurrent execution, and second, a more efficient, special-purpose, sequential execution environment as a node in a parallel environment.

The objective of this research is the design and performance evaluation of a special-purpose architecture for the efficient execution of FCP. Our primary research goal is to investigate a special-purpose single processor architecture as a step towards a parallel multiprocessor architecture. The objective is to show ways of enhancing system performance compared to previously proposed processor architectures for the execution of FCP. We are particularly concerned with the design approach used to propose the architecture and the methodology for performance evaluation.

An important part in the design stage is the analysis of previously reported and suggested implementation bottlenecks. The most significant of these bottlenecks is the overhead of goal suspension and activation, together with other goal management operations, such as goal creation and termination. The analysis should provide insight into the design of the special-purpose architecture.

The field of performance evaluation of computer systems is a well established and documented area of research [Ferr78], [Svob76], [Lave83]. However, this field is very poorly represented in the evaluation of logic programming systems. Our objective is to contribute with a more systematic and meaningful approach to performance analysis and evaluation of the proposed special-purpose processor architecture for FCP.

1.3 Contributions

We partition the contributions of this thesis into the following three areas:

1. Performance analysis of existing FCP implementations and design approach.

2. Special-purpose FCP processor architecture.
3. Performance evaluation of the proposed FCP processor architecture.

Performance Analysis

Since the motivation for the analysis of existing general-purpose implementations of FCP is to propose a special-purpose architecture, we claim that the analysis results must provide information that is independent of the existing implementation. We propose high-level analysis that captures the algorithmic behavior of FCP program execution.

To obtain the high-level analysis results, an instrumented version of the existing Logix emulator for FCP was written called Statistics Logix or *Slogix*. Measurements are obtained using existing applications written in FCP.

During the process of performance analysis, we use a real workload that is representative of the existing development environment. The workload consists of large FCP applications such as simulators, compilers, the Logix Operating System etc. We refer to this workload as the System's Development Workload.

The performance analysis is based on a set of analytic performance models. They are used to describe the relative execution time of previously suggested implementation bottlenecks. The performance models distinguish high-level program parameters from low-level implementation dependent parameters. The high-level algorithmic parameters are derived empirically by characterizing the System's Development Workload. The analysis is then performed for a range of implementation dependent parameter values. The individual performance models are then combined into one general model for the average execution time of a goal reduction.

From the performed analysis, we conclude that the overhead of goal management as well as the overhead due to *redundant* clause selection are the two main implementation bottlenecks. In addition we model the relative execution time of argument dereferencing and data trailing. The overhead of clause selection can be improved with better compilation techniques. This feature is also considered in the analytic models. However, reducing the overhead of clause selection using for example, clause indexing techniques results in the increased relative execution time of goal management operations. These results strongly motivate the need for architectural support for goal management. A more detailed collection of the

high-level analysis results is described in [Alka89].

FCP Special-Purpose Processor Architecture

We propose a special-purpose processor architecture for the efficient execution of FCP [Alka88]. The processor consists of multiple functional units that operate concurrently. The Reduction Unit executes a RISC instruction set with specialized instructions to support FCP execution. Other functional units include the Goal Management Unit, Tag Unit and the Instruction Unit. In addition, specialized cache units are defined for data, goals and instructions.

The main feature of the processor control mechanism is the overlapped execution of goal management, performed by the Goal Management Unit, and goal reduction performed by the Reduction Unit. By defining data structures to support the execution model, the two operations are completely decoupled thus enabling the overlapped mode of operation.

The efficient execution of goal management operations is performed using a special-purpose Goal Cache. It acts as a buffer of recently spawned goals. Goals are always scheduled for execution from the Goal Cache. We define the Goal Cache policy which consists of algorithms to perform goal creation, termination, suspension and activation. Switching between goals in the Goal Cache is performed without delaying goal reduction by having the Instruction Unit prefetch the first instruction from the next scheduled goal in the Goal Cache.

The processor architecture defines support for data trailing as part of the special-purpose Data Cache. The Data Cache policy called *Delayed Binding*, marks all assignments during a clause-try as *temporary* until the outcome of the clause-try is known. If the clause-try commits, the bindings become *permanent*. Otherwise, they are *cleared*.

FCP Processor Evaluation

To evaluate the main feature of the FCP processor architecture, namely the overlapped execution of goal reduction and goal management, we define a set of performance measures and analytic performance models. We conclude that, for the System's Development Workload, the overhead of goal management is reduced from approximately 50% of the program execution time when there is

no architectural support for goal management, to less than 3% of the program execution time when the overlapped Goal Management Unit is used.

We compare the proposed FCP processor architecture to the performance of other existing special-purpose processor architectures for FCP. For the same workload, we conclude that a speedup of at least 2.5 can be achieved by combining the speedup due to the Goal Management Unit and the Data Cache, with respect to the previously proposed processor, [Hars88].

We analyze the performance of the special-purpose FCP processor for a different set of workloads. The System's Development Workload that we use, exhibits both a higher complexity of goal management and a higher granularity of goal reduction compared to previously reported benchmark applications. Other application areas such as modeling communication protocols or distributed algorithms exhibit a similar complexity of goal management but a lower granularity of goal reduction. For these type of workloads, the attainable speedup due to overlapped execution of goal management with goal reduction is proportional to the ratio of their relative execution times.

Finally, we contribute by showing how the FCP processor architecture is used in a shared-memory multiprocessor environment. The use of the private Goal Management Units and the corresponding Goal Caches act as local pools of tasks or goals. Each Goal Management Unit first executes goals efficiently in the local Goal Cache. If the Goal Cache overflows, goals are shared in the common goal pool stored in the Goal Memory. Load balancing of goals is thus implicitly defined and also performed in an overlapped mode of operation, while the Reduction Units are busy.

1.4 Outline of Dissertation

This dissertation is organized as follows. In Chapter 2 we define the execution model of FCP. We do so by first discussing the main features of logic programming languages in general. The focus is on the various computation models. We specifically emphasize the transformations of the AND/OR computation tree, starting from Horn Clause logic programming languages and then describing the class of committed-choice, Guarded Horn Clause languages, such as FCP.

In Chapter 3, we outline the related work. The closest research to our own is the design of the RISC processor for FCP [Hars88]. We compare our work to

special-purpose Prolog processors, parallel inference machines from the Institute for Fifth Generation Computer Technology, ICOT, and parallel shared-memory implementations of logic programming languages.

In Chapter 4, we describe the previous implementations of FCP on general-purpose machines. This is important since many of the implementation features also apply to the FCP processor implementation. Particularly, the sequential abstract machine for FCP is modified to derive the special-purpose processor execution model.

In Chapter 5, we define the method for the analysis of existing implementations, and the design approach. This chapter consists of two main sections. First, we describe the tradeoffs of the analysis and we define the System's Development Workload used for empirical performance analysis. In the second part, we present the analytic models and the results of analysis. We also describe a general model of FCP goal reduction, that combines all the individual models together.

In Chapter 6, we define the architecture of the special-purpose processor. It is proposed based on the results of analysis described in Chapter 5. We describe the processor organization and execution model. The processor consists of multiple functional units that execute concurrently on a single processor. The processor is hierarchically structured with execution units accessing separate memory sections using specialized cache units. The two main execution units are the Reduction Unit and the Goal Management Unit, and the specialized caches are the Data Cache and the Goal Cache.

In Chapter 7, we describe the partitioning of the sequential abstract machine for FCP into two decoupled and concurrent abstract machines. We describe the main algorithms for overlapped goal management and goal reduction. Moreover, we define the Goal Cache policy and how it is used to perform goal management operations efficiently.

In Chapter 8, we evaluate the overlapped execution of goal management operations. We propose a set of analytic models and we perform the analysis using the specific implementation parameters according to the processor organization specified in Chapter 6, and the program parameters obtained in Chapter 5.

In Chapter 9, we evaluate the overall performance of the FCP processor. We also compare its execution to other existing processor architectures and we evaluate the contribution of the overlapped goal management strategy when applied to different system workloads.

Finally, in Chapter 10, we conclude our thesis and discuss extensions of the processor architecture to a multiprocessor environment. We also show how the load balancing of goals is performed using the overlapped goal management units.

CHAPTER 2

Concurrent Logic Programming Languages

In this chapter we focus on the description of the execution model of concurrent logic programming languages. However, this description is incomplete without a historic perspective and comparison to some of the language predecessors such as Prolog proposed by Colmerauer or Parlog [Clar86]. This section is organized as follows. First we describe the main features of logic programming in general followed by a description of the first practical logic programming language called Prolog. Various parallel models for the execution of logic programming languages are reviewed. This then introduces the concept of *concurrent logic programming*, which is distinguished from the term *parallel logic programming*. The differences between the two concepts is established and finally a simple abstract execution model for FCP, a concurrent logic programming language, is described.

2.1 Logic Programming

Using first order predicate calculus as the basis for programming languages has been the theme of a number of recently published books [Lloy84], [Kowa79], [Brat86], [Ster86]. In the following, we briefly present an outline of the main features that are important for the reading of this thesis. For more detailed information, the reader should consult the referenced books.

Horn Clause Syntax

A logic program \mathcal{P} can be represented as a set of sentences or clauses of the form:

$$\mathcal{P} = \{S \mid S = \forall_i(X_i)(H \leftarrow B_1, B_2, \dots, B_n), (n \geq 0)\}$$

where X_i denote universally quantified variables whose scope is the clause in which they are defined. In other words, in each clause, a distinct set of variables is assumed. Declaratively, the clause is read as an implication of conjunctive

a disjunction of implications. In other words, *either* the first clause rule for the mother relation is true (clause 2) *or* the second (clause 3). For simplicity, the universal quantifier in the clauses is commonly omitted and is implicit for all the variables in the clause.

Let us now consider the clauses of program \mathcal{P} and the following goal statement $\leftarrow \text{mother}(\text{Judy}, X)$, that is, the set $\{\mathcal{P}, \leftarrow \text{mother}(\text{Judy}, X)\}$. The goal statement is interpreted as the following query: “Given the program \mathcal{P} , does there exist an assignment to X such that $\text{mother}(\text{Judy}, X)$ is true in \mathcal{P} ?” Or simply, “Who are the children of Judy ?”

The literal *mother* in the goal statement or in the clause-body is referred to as a *procedure call*. Therefore, the second clause in the procedure for *mother* defines a recursive procedure.

Resolution and Unification

The procedure call execution consists of applying the rules of inference defined in the program to the arguments of the procedure call. The method of applying the inference rule is called the *resolution* principle [Robi65]. Given a goal statement or *resolvent* $\mathcal{R}: \leftarrow A_1, \dots, A_i, \dots, A_n$, a single resolution step consists of selecting a goal A_i from the resolvent, finding a matching clause $A'_i \leftarrow B_1, \dots, B_m$, and forming a new goal statement $\leftarrow (A_1, \dots, B_1, \dots, B_m, \dots, A_n)\theta$ where θ denotes the variable substitution set derived from matching arguments in the procedure call A_i with arguments in the clause-head A'_i . The variable substitution set, θ is a set of pairs $(V \rightarrow T)$ where V denotes variables and T terms such that all the variables are distinct and V s do not occur in T . Applying the substitution set θ to the resolvent \mathcal{R} means replacing every occurrence of V in \mathcal{R} with T for every pair $(V \rightarrow T) \in \theta$. For example, applying the substitution set $\theta = \{X/2, Y/f(a)\}$ to the term $g(X, p(Y), Y)$ results in the term:

$$\theta g(X, p(Y), Y) \Rightarrow g(2, p(f(a)), f(a))$$

In general, applying the substitution θ to term T results in $T' = T\theta$ where T' is called an *instance* of T .

The algorithm to find the set of variable substitutions that results in the most general common instance of two terms is called *unification*. The unification of terms T_1 and T_2 denoted $\text{unify}(T_1, T_2)$ results in the substitution θ such that $T_1\theta = T_2\theta$ if unification *succeeds*. Alternatively, the “not unifiable message” is

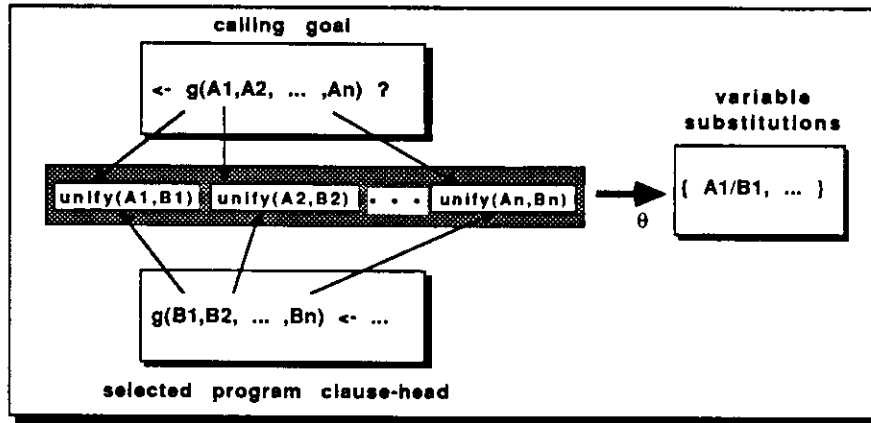


Figure 2.1: Goal-Head Unification

returned if unification *fails*. Unification of two atoms succeeds if they are both the same. Variable X unifies with a term T that does not contain the variable X and is denoted $\{X/T\}$. The unification of two complex terms is recursive. They unify if they denote the same functor (they have the same functor name and arity) and if all of their arguments unify.

In Figure 2.1 we show the general case where a goal $g(A_1, \dots, A_n)$ is being unified with the clause-head $g(B_1, \dots, B_n)$ producing a substitution set θ , where each element of θ is of the type $\{A_i/B_i\}$.

An Abstract Interpreter

The following is the algorithm of an *informal* abstract interpreter of a logic program \mathcal{P} and an initial goal *resolvent* G .

While the *Resolvent* is non-empty:
 Select a goal A from the *Resolvent*
 Select a clause $A' \leftarrow B_1, \dots, B_n$ from \mathcal{P}
 IF $\text{Unify}(A, A') = \theta$
 Replace A with B_1, \dots, B_n in *Resolvent*
 Apply θ to *Resolvent*

Successful program termination is interpreted as a proof procedure of the goal G in the program \mathcal{P} . However, the useful information is contained in the substitution set θ such that the instance of the original goal statement $G, G \theta$

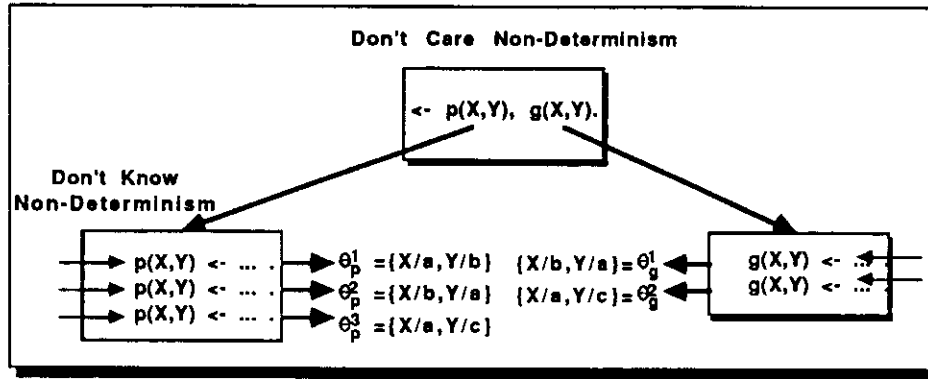


Figure 2.2: Don't Care and Don't Know Non-Determinism

is derived by the abstract interpreter. Therefore, θ is the composition of all the θ_i derived during the inference steps of the abstract interpreter that ended in an empty resolvent.

Goal Selection

The selection of the goal from the current goal resolvent is non-deterministic. In other words, one 'does not care' which goal selection approach is used. This type of non-determinism is referred to as *don't care non-determinism*. Moreover, since the goal statement is interpreted as a conjunction of goals, it is also referred to as *AND-Parallelism*.

Clause Selection

Each clause in a procedure represents an independent procedure call entry-point corresponding to separate inference rules. At procedure-call time, it is 'not known' which entry-points will lead to a *successful* solution and which entry-points will lead to *failure*. Moreover, more than one entry point may succeed, thus producing multiple solutions. The non-deterministic selection of inference rules used to resolve the procedure call is referred to as *don't know non-determinism*. Since the declarative reading of clauses in a procedure corresponds to disjunction, the non-deterministic selection of inference rules is also called *OR-Parallelism*.

In Figure 2.2 we show a goal statement that consists of two goals $p(X,Y)$

and $g(X,Y)$. If we assume that there are 3 clauses that denote the predicate relation p and 2 clauses that denote the g relation, then procedure p has 3 entry points and procedure g has 2 entry points. Attempting program execution at each entry point may result in the assignment of terms to variables using the previously described unification algorithm. In Figure 2.2 we show the case where the 3 inference rules corresponding to goal p produce 3 different sets of values for variables X and Y . The sets are labeled θ_p^1 , θ_p^2 and θ_p^3 . In a similar way, the two inference rules corresponding to goal g result in variable bindings sets θ_g^1 , and θ_g^2 .

Since the goal statement denotes the conjunction of goals, only those substitution sets that form a consistent set of bindings, when combined together, are considered as valid solutions. That is, conflicting sets of bindings are interpreted as failure. For example, if $X/a \in \theta_i$ and $X/b \in \theta_j$ then the attempted solution path that produces the bindings $(\theta_i \theta_j)$ that is, their conjunction results in failure. In other words, successful program execution of a goal statement results in a multiset of consistent variable substitutions, with each set being the result of applying one inference rule.

Therefore, alternative inference rules in the same procedure result in binding sets that correspond to multiple solutions whereas the conjunctive set of goals correspond to a set of *constraints* on the variable bindings belonging to only one solution. Only those substitution sets that satisfy all of the conjunctive constraints defined in the goal statement are considered as valid program solutions.

Logical Variables

The special treatment of the variable in logic programming deserves further attention, since it enables the use of many programming techniques that are unique to the logic programming model. In contrast to imperative languages where a variable represents a location in the program's run-time environment, the logical variable is used to *denote* objects in the logic program execution environment. Therefore, assigning the complex structure $f(a,b,c)$ to the logical variable X is interpreted as: "X denotes $f(a,b,c)$ ". Moreover, variable X can be assigned to another variable Y meaning that X and Y denote the same object.

The other feature that distinguishes the logical variable from variables in other programming languages, is the meaning of assigning a value to a variable. Given the set of variables in the logic program, successful program termination can be described as a multi-assignment of values to variables such that the assigned set

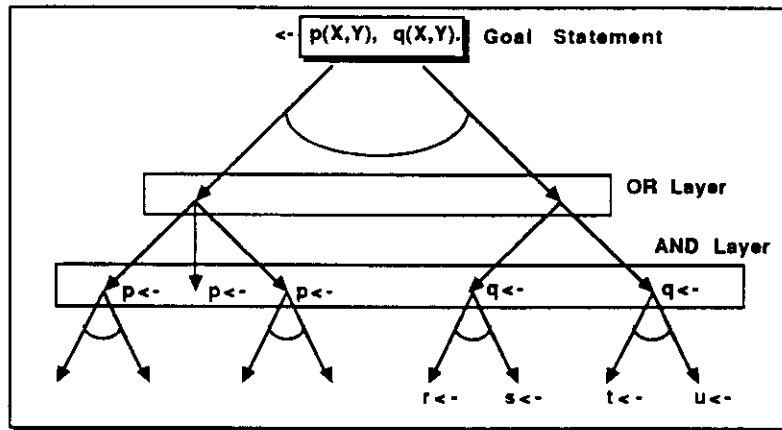


Figure 2.3: AND/OR Computation Tree of a Logic Program

can be derived by applying inference rules defined in the program. Therefore, during a single proof procedure, variables are defined as strictly *single-assignment variables*. Once a value is assigned to a variable, all further references to the variable denote the value. An alternative value can be assigned to the same variable only if another proof procedure is considered, in which case the previous value is revoked.

AND/OR Computation Tree

The computation of a logic program can be described using an AND/OR tree. In Figure 2.3 we show the tree structure for the following goal statement $\leftarrow p(X,Y), q(X,Y)$ and a simple program. The tree contains two sets of nodes: *AND* nodes and *OR* nodes which are interleaved, forming layers. The conjunctive relation between goals is denoted using a horizontal arc that interconnects the vertical arcs between two *OR* nodes. There is one *OR* node for each *AND* literal in the goal statement and one *AND* node connected to the *OR* node for each corresponding clause that matches the goal literal.

Program execution consists of traversing the AND/OR tree according to a predefined strategy. We now describe several AND-OR execution models that have been proposed.

2.2 Parallel AND-OR Execution Models

Several models for the parallel execution of the AND/OR computation tree have been proposed. They can be distinguished according to how they address the following two issues:

- The type of parallelism supported by the execution model.
- The way substitution sets that correspond to multiple solution paths are managed.

We briefly review several models. A more detailed description of the various parallel execution models is presented in [Cone87], [Wise86], [Ciep83], [Fagi87b].

A *pure* or full parallel execution model explores all solutions to a goal statement in parallel (OR-parallelism), and applies all conjunctive constraints in parallel (AND-parallelism). This may result in a combinatorial explosion of parallelism that is impossible to handle. This is particularly true for programs like chess or other games where for every conjunctive goal there are many disjunctive alternatives. Therefore, some mechanism for controlling the amount of parallelism in an AND/OR system is essential.

AND-OR Process Models

In common for all of the described AND-OR process models is the fact that they consider the complete AND/OR computation tree, computing more than a single solution, if available. We refer to these models as *Parallel Logic Programming Models* and we distinguish them from *Concurrent Logic Programming Models* that consider only parts of the AND/OR computation tree and find only single solution. We describe the Concurrent Logic Programming models later in this chapter. A review of the various implementations of different AND-OR models of computation is described in Chapter 3.

In the AND-OR process model proposed by Conery [Cone87], AND and OR processes are spawned corresponding to the nodes in the AND/OR tree. The process model pursues multiple paths in parallel, but computes only one solution at a time. The OR process is spawned for each literal in the goal statement. It, in turn, spawns AND processes for each clause that can potentially match

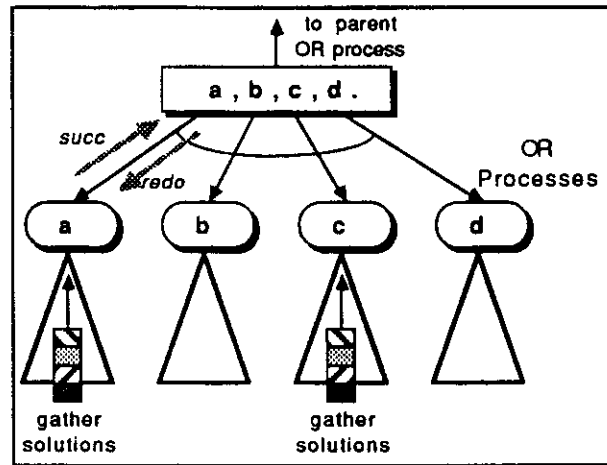


Figure 2.4: AND-OR Process Model

the selected literal. In Figure 2.4 we show an example where the AND process that consists of four goals spawns OR processes for each goal literal. Each OR process distinguishes the first successful variable substitution set from its child AND node, and reports the bindings to its parent node using the *success* message. All subsequent bindings sets from the child OR process are delayed until the parent demands them by sending the *redo* message. Meanwhile, the OR process runs in the *gathering* mode of operation. In this mode, all additional solutions that it may compute are stored in a list of solutions. They are given to the parent process only on request. Therefore, the OR parallelism is controlled by implementing a *demand driven* approach to finding multiple solutions.

The AND parallelism is controlled by executing in parallel only those goals that have no unbound variables, and thus cannot result in conflicting bindings. This approach is called *Restricted AND-parallelism* [DeGr84].

A more general approach for exploiting AND-parallelism in logic programs is described in [Lin88]. In this execution model, the conjunctive goals in a clause are ordered according to a *generator* and *consumer* relation using shared logical variables. The leftmost goal (including the clause head) that shares variable X is selected as the generator, and the remaining goals in the clause that share the variable, are marked as consumers. When a goal executes and produces bindings, it sends tokens to the consumer goals. When a goal becomes the generator of all the variables in its local environment, only then is it scheduled for execution.

Besides maintaining a list of tokens that indicate the order in which the goal

processes should execute, called *forward execution*, each process stores in a list, the names of processes that generate the bindings. In case of failure, a *backward* execution algorithm is used to request from the producer of the binding, an alternative solution.

Allowing the user to specify primitives that control the amount of parallelism and direct execution, is another approach. In the execution model described in [Chan85], the programmer declares the *activation modes* for each user predicate using three types of variable bindings: *non-ground and dependent*, *non-ground and independent* and *ground*. Using the activation modes, a static dependency-graph is generated at compile time. This graph is then used to schedule goals during forward execution.

In the AND-OR process model defined in Epilog [Wise86], sequential and and or primitives called CAND and COR respectively are defined, as well as *input/output* modes and *threshold* primitives to implement data-flow synchronization of conjunctive goals. The sequential operators are used to prevent the default creation of parallel AND and OR processes.

In the PEPSys [Baro88a] execution model, the degree of OR-parallelism and Restricted AND-parallelism is specified by the user using procedure declarations.

The main problem in exploiting OR-parallelism is how to maintain the multiple bindings of the same set of variables resulting from different solutions. Several approaches have been proposed. In the SRI model [Warr87], binding arrays are associated with each shared variable and in the PEPSys model hash windows are used. A good analysis of the different approaches and the tradeoffs involved is described in [Warr87].

The tradeoff between the different AND-OR execution models that consider the complete AND/OR computation tree, is the degree of parallelism versus the complexity of the model.

2.3 Sequential AND-OR Execution Model

One way of *controlling* the amount of parallelism in the AND/OR computation tree is to execute it sequentially. This actually resulted in the first efficient implementation of the logic programming model of computation and in the first logic programming language, called Prolog. We now discuss the Prolog execution

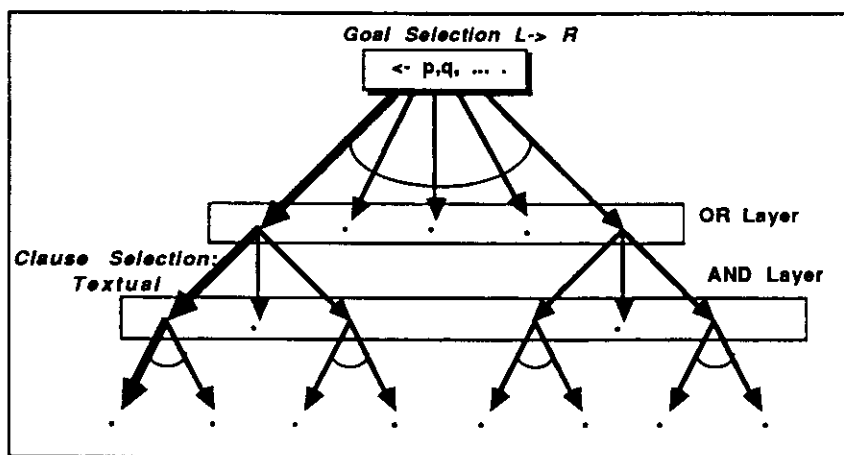


Figure 2.5: Prolog Computation Tree: Sequential Search

model in more detail, since it also lead to the development of the Parallel Prolog execution model subsequently discussed.

Prolog

Prolog is the first widely used logic programming language that executes the AND/OR computation tree in a sequential manner. In Prolog, goals in the goal statement are selected from left to right and the corresponding inference rules are applied in textual order. This corresponds to a *depth-first* search strategy, as highlighted in Figure 2.5.

If an inconsistent binding set is made during the application of an inference rule, the search path fails and the last alternative search path is followed. This process of *rescinding* an inference rule that was previously applied and applying a new rule is referred to as *backtracking*. If all search paths fail and there is no alternative path to search, the program fails.

The sequential, depth-first execution strategy of Prolog has the following very important property which enabled its wider use: an efficient stack-based abstract machine implementation was defined by Warren [Warr83]. Given the tree structure of computation, Prolog execution using the sequential abstract machine resembles the way procedure records are pushed onto a stack in imperative languages. Upon failure, Prolog efficiently pops the records on top of the stack until an alternative path (called a choice-point) is found.

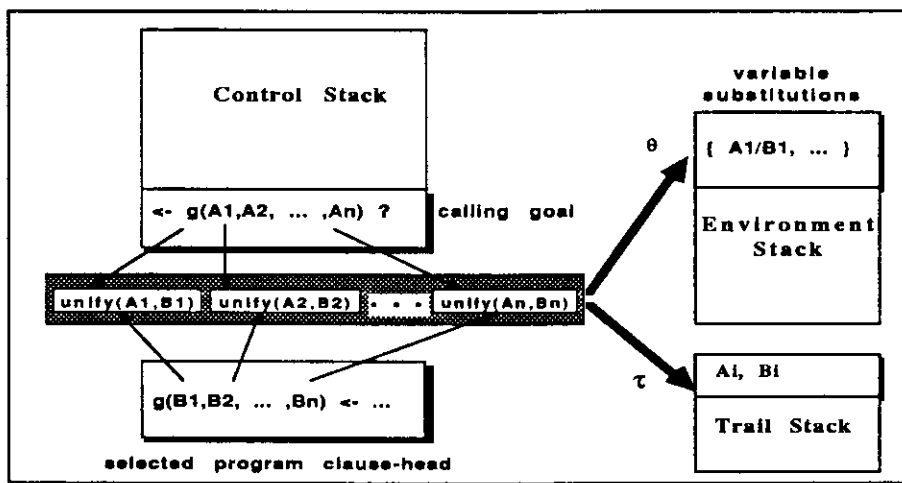


Figure 2.6: Prolog Clause-Try

Similarly, the binding sets produced during forward execution are stored using a stack of environments. Upon failure, backtracking through environments follows the backtracking of control. Moreover, all of the bindings created during the failed forward execution must have been trailed using a trail stack. Upon backtracking through control and environments, all of the bindings are undone.

In Figure 2.6 we show the unification of a left most goal from the current goal statement stored in the control stack, with a matching clause-head defined in the program. Besides the variable substitution set θ which is recorded in the current environment allocated for the goal, the set of addresses τ of variables that have received values are stored in the trail stack.

Therefore, it is the multiple stack architecture of the Warren Abstract Machine (WAM) that resulted in an efficient implementation on general-purpose processors.

2.4 Parallel Prolog Execution Model

Because of the fine granularity of processes as well as the complexity of the inter-process communication protocols, the AND-OR process models that eagerly exploit concurrency are outperformed by the sequential execution of the AND/OR tree using the WAM. This fact resulted in the following approach we refer to as the *Parallel Prolog* execution model. Program execution consists of

parallel sessions of sequential Prolog execution. Given the AND/OR tree, a single Prolog session starts from the root of the tree. Given the availability of parallel processing elements, an OR branch in the AND/OR tree is partitioned and sent to the idle processor. All busy processing elements continue to execute sequential Prolog.

An example of a Parallel Prolog system is the Kabu-Wake method [Sohm85], that partitions the tree in half at a point closest to the root, in order to increase the granularity of the spawned Prolog sessions. The main overhead is introduced when the tree is being partitioned, since it requires the management of shared variables that are sent together with the partitioned AND/OR tree.

Another approach is defined in the Aurora system [Lusk88], which subsumes the approach of Kabu-Wake. In Aurora, parallel execution depends on the availability of parallel *workers* (processors or parallel processes). Available workers *search* for work by *walking* up and down the computation tree. In Chapter 3, we discuss in more detail both the Kabu-Wake and the Aurora Parallel Prolog implementations.

2.5 Concurrent Logic Programming

Concurrent logic programming languages are parallel programming languages that have evolved from the abstract logic programming model. Syntactically, the main difference compared to Horn Clause logic programs is the addition of the *commit* control operator $|$, which partitions the clause into the *guard* part and the clause-body. A *guarded horn clause* is represented as follows:

$$\mathcal{P} = \{S \mid S = \forall_i(X_i)(H \leftarrow G_1, \dots, G_m \mid B_1, B_2, \dots, B_n), (n, m \geq 0)\}$$

where the G_i s represent guard goals. The clause is read as an implication of conjunctive guard goals and body goals. Semantically, the main departure from the non-guarded horn clause reading is that the application of an inference step commits to the body of at most one clause. Given a literal in the goal statement and several matching guarded clauses, the body of only one clause is used for continued program execution. Program execution *commits* to the body of the clause whose goal-head unifies with the selected goal and whose guard goals evaluate successfully. If the clauses are evaluated in parallel, after commitment to one clause, all alternative clauses are disregarded.

The use of the commit operator resembles the use of the *cut* control mecha-

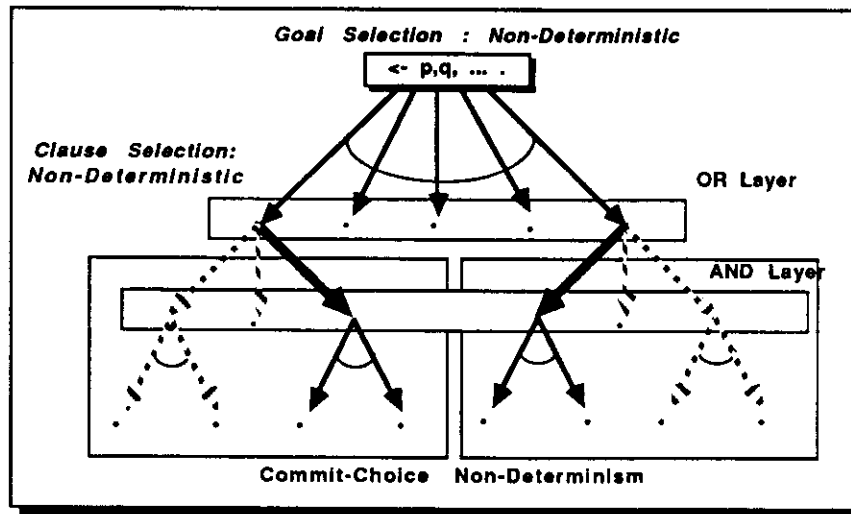


Figure 2.7: Guarded Horn Clause Computation Tree

nism used in Prolog to prune the control stack when it is clear that a deterministic path is being explored. Using the commit operator, program execution does not backtrack across committed boundaries.

Therefore, concurrent logic programming languages do not implicitly search for multiple solutions of a goal statement. If this is required, it must be explicitly programmed by the user. This is a significant departure from the AND/OR semantics of logic programs and defines a different application domain compared to programs that consider the search for multiple solutions an essential part of the abstract computational model.

In Figure 2.7 we show the computation tree of a guarded horn clause logic program. Since the goals in the guard may themselves spawn additional guarded goals, a nesting of alternative paths may be created in the same way as it was done in the non-guarded clauses. However, at each level, a commit operation results in the elimination of part of the computation tree thus resulting in a deterministic path to the solution. This class of languages are also referred to as *committed-choice* languages. The *bounded OR parallelism* is achieved by having goals in the guard.

Committed-Choice AND Parallelism

The main type of parallelism used in concurrent logic programming languages is AND parallelism rather than OR. Using AND parallelism in non-guarded horn clause programs created the problem of synchronizing access of conjunctive goals to shared variables. Using guarded clauses, program execution commits to a single binding set that belongs to only one solution. Since committed-choice AND parallelism is the main form of concurrency in this model, the question remains which conjunctive goal will produce the bindings of a shared logical variable if two goals execute concurrently. The issue of accessing shared logical variables is addressed by defining *rules of suspension*. We now discuss approaches that distinguish three programming languages.

Rules of Suspension

Rules of suspension in concurrent logic programming languages define the mechanism used to synchronize access to logical variables shared amongst conjunctive goals. Without rules of suspension, the behavior of two goals that share a logical variable depends on which goal commits first, setting the binding to the variable. To avoid this erratic behavior, the language Concurrent Prolog [Shap83] supports user annotations of shared logical variables. The user may annotate a shared variable X as read-only, using the *read-only* annotation $?$, that is, $X?$. A goal that attempts to assign a value to the read-only version of the variable will suspend waiting for variable X to receive a value. Presumably, another goal contains the writable occurrence of the same variable and will eventually assign the value.

In the following example, two goal predicates p and q share the same logical variable X . Goal p has read access while goal q write access. The program clauses for the two predicates both assign the value a to X as follows:

$$\begin{aligned} \text{goal statement : } & \leftarrow p(X?), q(X). \\ \text{clause for } p : & p(Y) \leftarrow Y = a \mid \dots \\ \text{clause for } q : & q(Z) \leftarrow Z = a \mid \dots \end{aligned}$$

The same semantics of program execution should be maintained regardless of the order in which the two goals p and q execute. If goal p is scheduled first, it will attempt to assign the value a to the variable Y where the substitution $\{X?/Y\}$ results from goal-head unification. Since the goal suspension rule does not allow

a value to be assigned to a read-only variable, the combined substitution set $\{X?/Y, Y/a\}$ results in the suspension of goal p waiting for X to receive a value. When goal q gets scheduled, the goal-head unification and guard evaluation results in the substitution set $\theta = \{X/Y, Y/a\}$ which does not suspend. After this, the goal p can be scheduled for execution. Therefore, using the *read-only* annotations of variables, one can model the direction of unification imposing an ordering of goal reduction.

In the programming language Parlog [Clar86] the user uses procedure input/output modes to indicate the directionality of arguments defined in the clause-heads. At compile time, it is determined whether the argument should have a value or not. Using the same example as above, if the variable argument X is declared as input using the \downarrow *input mode* declaration, then, if the argument is an unbound variable at run-time, the goal will suspend until the value for the variable becomes available. In other words, the two predicates are declared as follows:

goal statement : $\leftarrow p(X), q(X)$.
declaration for p : $p(X \downarrow)$.
declaration for q : $q(X \uparrow)$.
clause for p : $p(Y) \leftarrow Y = a \mid \dots$
clause for q : $q(Z) \leftarrow Z = a \mid \dots$

The programming language called Guarded Horn Clause or simply GHC [Ueda86] defines implicit goal synchronization that does not require any user annotations of variables or input/output declarations. The following two suspension rules are defined in GHC. First, a goal suspends if an attempt is made to bind a variable in the clause-head or guard. Assigning values to variables can only be performed in the clause-body. The second rule is that the body goals cannot assign a value to a variable before program execution commits to the body of the clause. This allows for a less restrictive semantics enabling body goals to execute before guard goals, as long as they do not modify the common environment. Using the same program, in GHC the consumer goal will check in the clause-head or guard whether the variable has a value. If not it will suspend. The producer of the value assigns the value in the body of the clause rather than in the guard.

goal statement : $\leftarrow p(X), q(X)$.
clause for p : $p(Y) \leftarrow Y = a \mid \dots$
clause for q : $q(Z) \leftarrow true \mid Z = a, \dots$

A fourth goal suspension mechanism is proposed by Saraswat in [Sara89] and

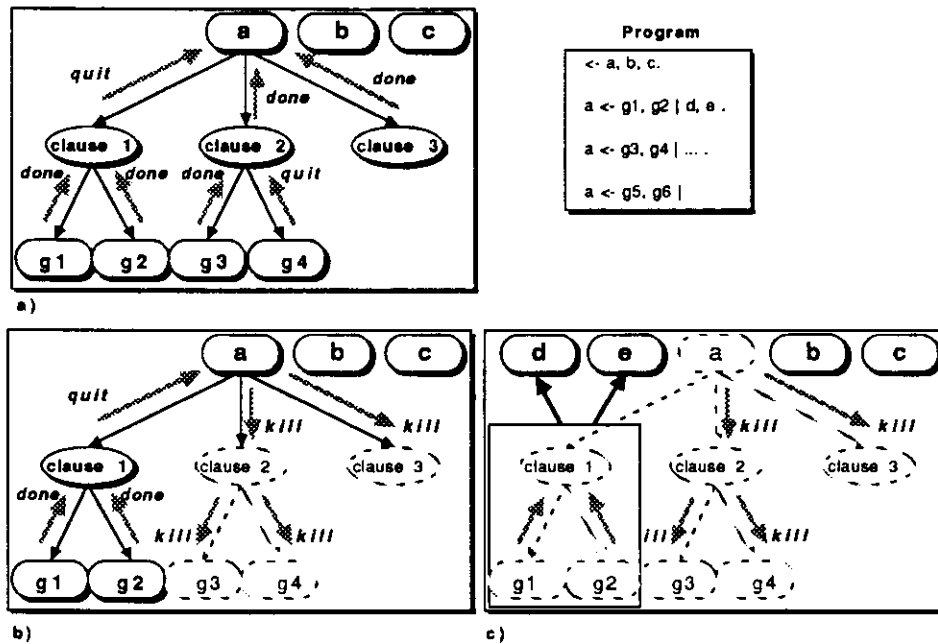


Figure 2.8: Committed-Choice Process Model

used in the programming language FCP(|,;,?) [Klig88b]. Here, the clause-guard is divided into the *ask* and *tell* part delimited by the : operator. In the *ask* part, implicit goal suspension is used if an assignment is attempted. This is like the mechanism used in GHC. Assignments are allowed in the *tell* part of the guard and in the body of the clause. In addition, *read-only* variables can be used.

2.5.1 Committed-Choice Process Model

A process model for the execution of committed-choice languages is defined by Crammond in [Cram86]. In this model, two types of processes are created: *goal processes* (corresponding to OR processes in the AND/OR tree) and *clause processes* (corresponding to AND processes). The processes communicate using three types of messages: *done*, *quit*, and *kill*. The model is best explained using a simple example shown in Figure 2.8.

Three goal processes are created for each literal in the goal statement. Let us consider the execution of the goal literal *a* that has three matching clauses in the user program. The goal process for *a* thus creates three *clause processes*. Intuitively, due to the committed-choice execution model, only one clause process

should be allowed to succeed, whereas all the other clause processes should be aborted (if they do not fail).

The clause processes attempt to unify the goal head with the corresponding clause head. Let us assume that the first two clauses succeed in doing so, and continue execution by spawning each two new goal processes corresponding to the clause guards. The third clause process, however does not succeed to unify the calling goal with the clause-head. This clause process sends a *done* message to the parent goal process, whereas the others continue with execution.

In Figure 2.8(a) we show the case where both guard goals g_1 and g_2 belonging to the first clause process succeed. When a goal process succeeds, it notifies the parent clause process by sending a *done* message. Alternatively, the two guard goals g_3 and g_4 belonging to the second clause process result in one success and one failure of execution. The failed goal process sends a *quit* message to the parent process whereas the successful goal sends the *done* message. Therefore, the first clause process receives two *done* messages denoting successful termination of its two guard goals, and the second clause receives one message denoting that one guard failed. As a result, the first clause process reports success to its parent by sending a *quit* message, and the second clause reports failure by sending a *done* message.

In response, the first clause process attempts to commit to the parent goal process by checking to see if there was any other clause that previously committed. Since there was not, the parent goal process is reduced to the body of the committed clause, and the parent process sends to all the other children clause processes *kill* messages to terminate their execution. The *kill* messages are propagated to all the child processes. This situation is shown in Figure 2.8(b).

Note, that the result of the committed-choice execution, besides terminating parts of the alternative branches in the goal/clause tree, the committed path is reduced to the body of the selected clause. This is shown in Figure 2.8(c).

2.6 Flat Concurrent Logic Programming Languages

Flat concurrent logic programming languages are a subset of concurrent logic programming languages that allow only simple predefined language primitives in the clause-guard. Examples of flat concurrent logic programming languages are Flat Concurrent Prolog (FCP) [Mier85], Flat Parlog and Flat GHC (FGHC).

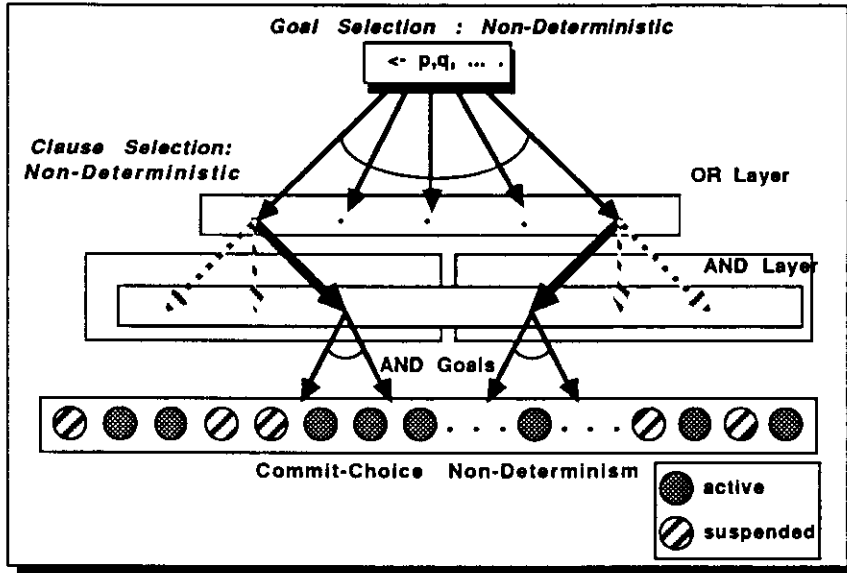


Figure 2.9: Flat Guarded Horn Clause Computation Tree

By restricting the guard goals to only simple primitive operations, a guard goal cannot spawn additional goals and therefore cannot result in the nesting of environments that need to be managed prior to commit. Some of the problems regarding the management of the non-flat environments in the guards of Concurrent Prolog are reported in [Sara87]. These problems are avoided by keeping the goal execution environment *flat*. The allowed OR parallelism is thus further constrained in flat CLP languages to only one level. We refer to this parallelism as *shallow OR parallelism* as opposed to *bounded OR parallelism* in non-flat committed-choice languages, and *full OR parallelism* in parallel logic programming languages. Within this single level of alternative clauses, a choice must be made, which clause to commit to based on the clause-guards and the arguments in the clause-head.

In Figure 2.9 we show the computation tree of flat concurrent logic programming languages. One should note that what remains from the original AND/OR computation tree of logic programs are those leaves that correspond to conjunctive goals. These goals could be scheduled for reduction in a non-deterministic manner and are synchronized using goal suspension rules. Since the various flat concurrent logic programming languages have their subtle differences, we now describe in more detail the execution model of Flat Concurrent Prolog. We will refer to this execution model throughout this thesis.

2.7 An Execution Model of Flat Concurrent Prolog

FCP program execution consists of non-deterministic goal reduction of conjunctive goals that correspond to the leaves of the computation tree shown in Figure 2.9. The reducible set of goals represents the *program resolvent*. The goals in the resolvent share logical variables that are used for inter-goal communication and synchronization. Also shown in Figure 2.9 are some conjunctive goals that were scheduled for reduction before a shared logical variable received a value. These goals are labeled as suspended.

We partition the resolvent \mathcal{R} into two disjoint sets: the active set of goals \mathcal{A} and the set of suspended goals \mathcal{S} . Goal reduction is performed by selecting and scheduling goals from the active set of goals.

The scheduled goal represents a unit of concurrent work in a system of conjunctive goals. It consists of a program and a finite number of goal arguments. The program denotes the set of control primitives (machine instructions) representing the program. Goal reduction consists of interpreting the control primitives using a defined interpretation algorithm. In Figure 2.10 we show goal g that has 3 arguments, represented as $g(A_1, A_2, A_3)$ and denoted using argument pointers. There are three possible outcomes of a goal reduction:

1. Goal-Failure
2. Goal-Commit
3. Goal-Suspension

Program execution requires all goals to succeed. Goal-failure results in program failure. However, detecting program failure can be implemented at a higher, meta-interpreter level. Goal-commit occurs when the selected goal unifies with one of the program clause-heads and all of the guards in the clause evaluate successfully. This then leads to goal reduction which will be described shortly. Goal reduction that requires additional data in order to determine goal-commit or goal-failure, results in goal-suspension. In Figure 2.10 we show the program resolvent partitioned into the active and suspended set and the non-deterministic selection of a goal scheduled for reduction.

Informally, the FCP execution model can be described as follows.

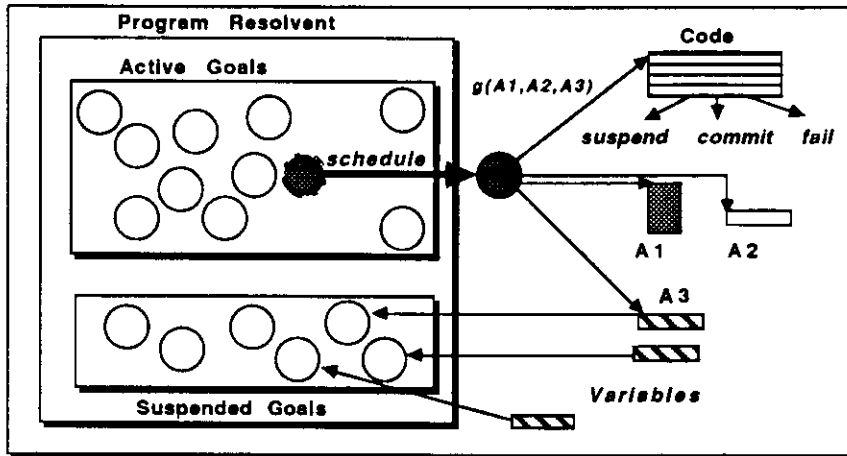


Figure 2.10: Program Resolvent: Non-Deterministic Goal Reduction

- While the set of active goals \mathcal{A} is not empty, for every $G_i \in \mathcal{A}$ do the following:
 - Attempt to find a clause whose head unifies with G_i and whose guard goals evaluate successfully, resulting in the variable substitution set θ . If successful, commit to this clause and replace G_i with the goals in the clause-body, and apply θ to the new resolvent. Also, add to the active set of goals \mathcal{A} all suspended goals in \mathcal{S} that have received values during clause-head unification.
 - If all attempts to find a matching clause-head *fail*, then fail.
 - Otherwise, suspend G_i on variables whose values eventually determine whether the goal will commit or fail. Goal G_i is moved to the suspension set \mathcal{S} .

Whether a goal fails, suspends or commits is determined by inspection of the goal arguments and matching them to one of the program clauses. An attempt to unify the calling goal with one clause is referred to as a *clause-try*. Therefore, the outcome of a goal reduction is determined by the alternative clause-tries. We now discuss in more detail the operations of a clause-try followed by a description of goal suspension and goal reduction.

2.7.1 FCP Clause-Try

In Figure 2.11 we show a clause-try performed in FCP. There are three possible outcomes of a clause-try:

1. Clause-Failure
2. Clause-Commit
3. Clause-Suspension

In the execution model for FCP considered in this thesis, we assume the sequential execution of clause-tries. In other words, we do not consider the parallelism of alternative clause-tries. If a clause-try fails, another clause-try is attempted until either a clause-try commits or suspends. If all clause-tries fail, the goal reduction fails. The sequential execution of clause-tries is considered for the following reasons:

- In FCP, because of the *flat* nature of the clause guards, the operations performed during a clause-try are generally simple operations. We assume that the penalty of spawning these primitive operations and executing them in parallel is greater than the performance improvement due to parallel execution.
- Performing a clause-try consists of matching arguments of a calling goal with arguments in the program clause. Reducing the execution time of a clause-try is an active area of research in the field of advanced compiler technology. For example, clause-indexing techniques exist to determine which clause is applicable to the calling goal. If more information is provided through argument type declarations or input/output modes, the clause-try execution time can further be reduced, and thus result in efficient sequential execution.

As shown in Figure 2.11, a clause-try in FCP is different from the clause-try in Prolog in the following three ways. First, in addition to producing the substitution set θ and the trail set τ , a clause-try in FCP records in a Suspension Variable Table (SVT) a suspension variable set s . During a clause-try, the addresses of those unbound variables where a value was expected, are stored in SVT. If the clause-try does not fail, and there is at least one element in SVT, then the outcome of the clause-try is *clause-suspension*. In other words, the clause-try did

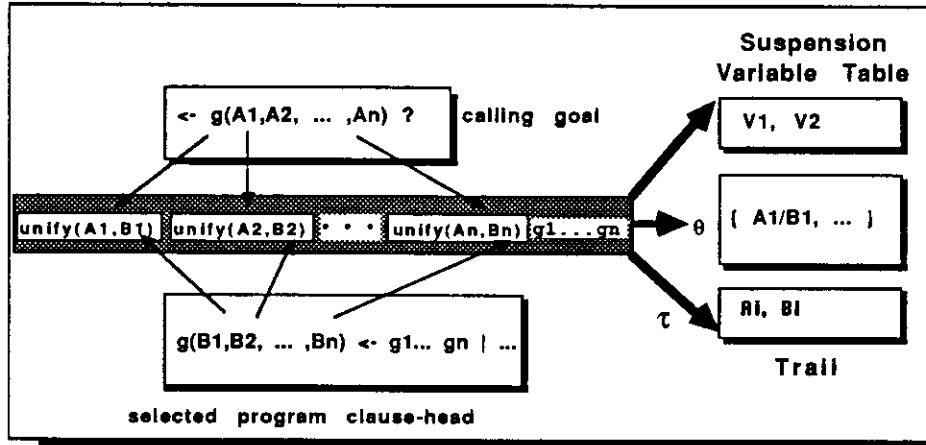


Figure 2.11: FCP Clause-Try

not commit or fail, but may succeed in the future when more data corresponding to variables in s become available. Second, the clause-try in FCP is augmented with primitive guard tests that also have to succeed prior to clause-commit. The commit operation prevents program execution from backtracking. Finally, the clause-tries in FCP can be performed in any order, as opposed to textual order of Prolog.

When a clause-try fails, the bindings performed during the clause-try are revoked, in the same way as it is performed in Prolog. This type of backtracking within the same procedure is called *shallow backtracking*, as opposed to *deep backtracking* that transcends across nested binding environments. There is no deep backtracking in FCP due to its flat committed-choice execution model.

2.7.2 Goal Suspension and Activation

The committed-choice semantics of FCP implies that at least one clause in the procedure must *eventually* commit. If any sequential ordering is imposed on the clause-tries, then, a clause-suspension does not immediately result in the goal-suspension. In other words, a goal suspends only if there is no clause-try that can succeed, and at least one clause-try did not fail. This implies that the clause-try may succeed in the future. A goal reduction suspends if there is at least one clause-try that suspends and none of the clause-tries commit.

For example, let us consider the following two cases of a selected goal that

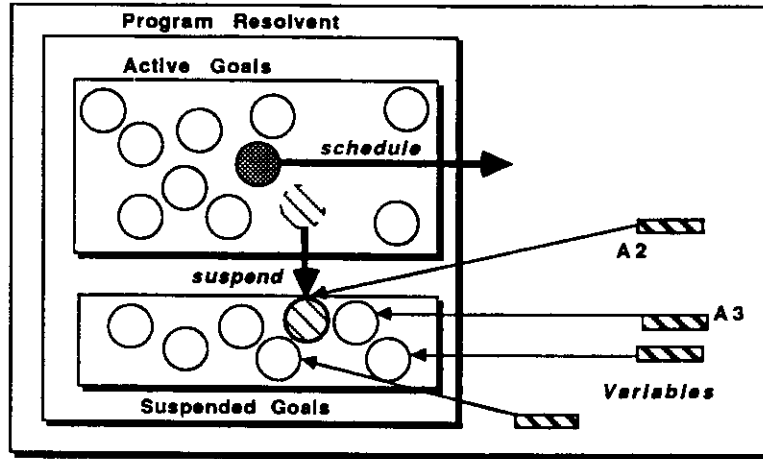


Figure 2.12: FCP Goal Suspension

has four potentially matching clauses in the program. If the first two clause-tries fail, the third suspends and the fourth commits, this results in goal-commit and the goal reduces to the body of the fourth clause. If, however, the first two clause-tries fail and the third and fourth suspend, the result is goal suspension using the variables stored in the suspension table.

Goal Suspension

Consider the program resolvent \mathcal{R} , the active set of goals \mathcal{A} and the set of suspended goals \mathcal{S} , such that, $(\mathcal{R} = \mathcal{A} \cup \mathcal{S} \text{ and } \mathcal{A} \cap \mathcal{S} = \emptyset)$. In addition, we label the set of all goals as $Goals$ and the set of distinct variables as $Vars$. We denote that goal $G_i \in Goals$ suspends on variables $\hat{V} \subseteq Var$ using the following relation: $suspend(G_i, \hat{V})$. With each variable $X_i \in Var$, a $Suspension_{list}$ function is associated that returns all goals suspended on X_i . The following steps are performed during goal suspension:

$$\mathcal{A} = \mathcal{A} \setminus G_i$$

$$\mathcal{S} = \mathcal{S} \cup G_i$$

$$\forall (X_i \in \hat{V}) : Suspension_{list}(X_i) = Suspension_{list}(X_i) \cup G_i$$

That is, the suspended goal is moved from the active set to the suspended set of goals and the goal G_i is associated with the *suspension list* of all the variables in the suspension set \hat{V} .

Goal Activation

Using the same notation as for the goal suspension, we denote the activation of a set of goals $\hat{G} \subseteq Goals$ by a set of variables $\hat{V} \subseteq Var$ as the following function: $\hat{G} = activation(\hat{V})$, where:

$$\hat{G} = \{G_i \mid G_i \in suspension_{list}(X_i) \wedge (X_i \cap \hat{V} \neq \emptyset)\}$$

In addition, the following operations are performed:

$$\mathcal{A} = \mathcal{A} \cup \hat{G}$$

$$\mathcal{S} = \mathcal{S} \setminus \hat{G}$$

$$\forall (X_i \in \hat{V}) : Suspension_{list}(X_i) = Suspension_{list}(X_i) \setminus G_i$$

In other words, the activated goals are moved to the active set of goals and the goals are removed from the suspension lists of the variables that activate the goals. By suspending when there is insufficient data to commit, and by activating the suspended goals when the data becomes available, a data-flow goal scheduling mechanism is implemented. In Figure 2.12 we show the case where a goal suspends after it is determined that the value of variable A_2 is required but unavailable during goal reduction.

Using the above description of goal suspension and activation, one should note the following. If goal G_i suspends on a variable X , it is associated with the suspension list of that variable. If the variable X is then unified with variable Y then variable X denotes variable Y by a substitution $\{X/Y\}$. If a clause-try results in the substitution set $\theta = \{Y/a\}$ then the goal G_i is activated since it is associated with the suspension list of X that is denoted by Y .

2.7.3 Goal Reduction

When it is determined that a clause-try succeeds with a variable substitution set θ , program execution commits to the selected clause and the selected goal is reduced to the body of the committed clause. All suspended goals that have received values in the substitution θ are rescheduled for execution, that is, activated. We distinguish the following two cases for a goal reduction. Either the committed clause-body is empty or it contains body goals. The fact that the

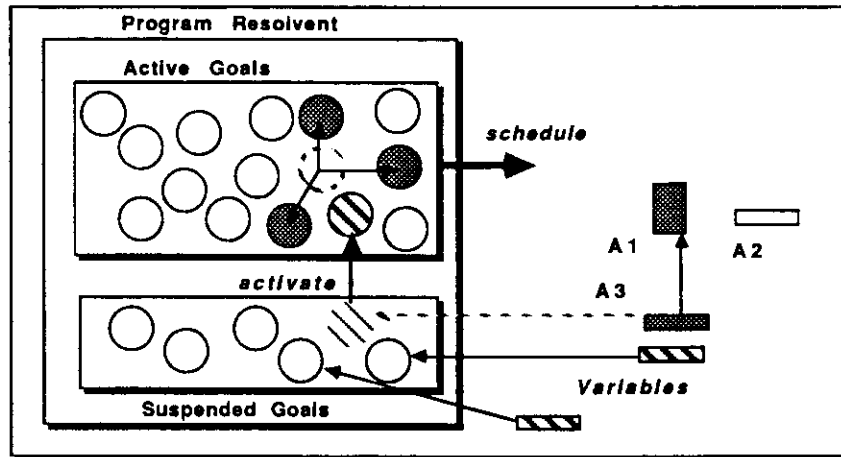


Figure 2.13: FCP Goal Reduction

empty clause-body is selected for reduction implies that the goal has successfully terminated and a new goal may be scheduled for reduction.

If the clause-body contains (sub)goals, these new goals replace the reduced goal in the program resolvent. In Figure 2.13 we show the case where the selected committed-clause reduced to three new goals; these are all added to the active set of goals from where they are scheduled for reduction.

2.8 FCP Programming Examples

A good survey of FCP programming techniques can be found in [Shap86] and in [Shap87]. We show here two simple examples of FCP programs, both from [Shap87]. The first is the stack monitor program and the second is the stream merge program. We briefly describe each in turn.

```

stack_monitor([push(X)|In],S) <-
    stack_monitor(In?,[X|S]).
stack_monitor([pop(X)|In],[X|S]) <-
    stack_monitor(In?,S).
stack_monitor([],[]).

```

The stack monitor interprets three types of messages on its input stream In

that modify the state of the stack S . If the `push(X)` message is received, the element X is placed in front of the stack S . This is denoted by managing the stack as a list, where X is the head of the list and S is the tail, denoted as $[X|S]$. When the `pop(X)` message is received, the element on top of the stack, X , is unified with the variable in the message. The stack monitor then iterates with the tail of the stack, S , thus denoting that the top of the stack was popped. If the null element is received on the input stream, the stack monitor terminates execution. Note that the read-only variable annotation of the input stream $In?$ prevents the monitor from writing onto the input stream. In other words, the monitor can only read messages sent to it.

In the `stream_merge` program, there are two input streams X and Y and one output stream Z . Messages from the input streams are merged onto the output streams. The first clause shows an element from the first stream copied to the output, and the second clause does the same for the second input stream. The streams are closed when the null element is received.

```

stream_merge([X|Xs],Y,[X|Z]) <-
    stream_merge(Xs?,Y,Z).
stream_merge(X,[Y|Ys],[Y|Z]) <-
    stream_merge(X,Ys?,Z).
stream_merge([],Y,Y).
stream_merge(X,[],X).

```

2.9 Chapter Summary

We have reviewed the computation model of non-guarded and guarded Horn clause programs by showing their relation to the AND/OR computation tree. In Flat Concurrent Logic Programming languages like FCP, only the conjunctive goals remain as leaves of the reduced computation tree and all the alternative OR branches are discarded during clause-commit. The only OR parallelism that remains is the clause selection parallelism, that is, *shallow OR parallelism*, which is not used in the FCP execution model, since all clause-tries are performed sequentially. This distinguishes CLP languages from non-committed choice languages that explore the full AND-OR model thus deriving multiple solutions to a given goal. It also defines a separate domain of program applications.

The computational model of FCP is described in terms of non-deterministic goal scheduling from a pool of active goals with a mechanism to implement data-

flow goal synchronization. The suspension rules in FCP differ from those of related CLP languages such as FGHC and Flat Parlog.

Process, Procedure and Co-Routine Model Analogy

In [Shap86] a process model analogy is drawn between the execution model of FCP goals and processes in a multiprocessing system. Goals are analogous to processes and the shared logical variables are analogous to the inter-process communication media. Goal suspension corresponds to process blocking and goal activation to resuming process execution. A goal reduction that commits to an empty clause is interpreted as process termination.

Another analogy is between goals and procedures. This analogy is already established for Prolog but can also apply to FCP. In fact, given the size and frequency of goal execution, the dynamic behavior of goals may be closer to procedure calls than processes in a multiprocessing system. However, the notion of goal suspension and activation resembles the use of *co-routine* operations such as *co-routine exit* and *co-routine resume*.

CHAPTER 3

Related Work

In this chapter we review previous work related to our research. We divide the description of proposed special-purpose architectures for logic programming languages into the following three categories:

- Special-purpose single-processor architectures.
- Parallel Inference Machines
- Parallel Inference Machines.
- Shared memory multiprocessor architectures.

In the first category we review the work that is most closely related to our research, that is, the design of a special-purpose processor architecture for FCP. We discuss the VLSI FCP processor design approach proposed by Harsat and Ginosar called *Carmel* [Hars88]. We also briefly review some of the special-purpose processors proposed for the execution of Prolog. Even though the Prolog execution model is significantly different from FCP, we discuss the common features that the two languages share as well as how these features are supported at the architectural level. As Prolog is a predecessor to FCP, the work on special-purpose architectures for Prolog have significantly influenced the design of architectures for committed-choice languages.

In the second category we consider parallel implementations of logic programming languages such as Concurrent Prolog, GHC, Parlog and Parallel Prolog. Five machine architectures proposed by the Japanese Institute for Fifth Generation Computer Technology (ICOT) are reviewed. Each machine consists of processing elements that are interconnected in a distributed environment with no shared memory. This research is related to our own in the following way. Initially, the parallel inference machines proposed by ICOT have embarked on implementing full Concurrent Prolog as the target language, as well as OR-parallel Prolog.

However, the complexity of the execution model as well as the difficulties encountered with the semantics of Concurrent Prolog, resulted in research for a simpler and more precise computation model, and thus languages such as FCP [Mier85].

In the third category we consider the more recent implementations of logic programming languages on general-purpose shared-memory multiprocessor architectures. These architectures are more recent, since the initial research conducted at ICOT proposed only distributed implementations with the intention of defining highly parallel systems. Since commercial general-purpose shared memory multiprocessors have become an affordable reality, they are being used to prototype parallel logic programming implementations.

First we review the implementation of Parlog, a committed-choice language, as described by Crammond in [Cram88]. We then describe the OR-parallel implementation of Prolog called Aurora [Lusk88]. Both Parlog and Aurora are implemented on the Balance Sequent multiprocessor. A detailed comparison of a committed-choice language KL1 (FGHC) versus the Aurora non-committed choice implementation on a shared memory multiprocessor is described by Tick in [Tick88]. We review the main results of this analysis. Whereas Aurora implements only OR-parallelism, we review two more parallel implementations of Prolog that allow for both OR and *Restricted* AND-parallelism. These are the PEPSys architecture proposed at ECRC [Baro88a] and the Aquarius project from Berkeley [Fagi87a].

3.1 Special-Purpose Single-Processor Architectures

Carmel VLSI Processor

A VLSI RISC processor for the execution of FCP is proposed by Harsat and Ginosar in [Hars88]. The processor data path is very simple and consists of a three-port register file with only 25 registers and one arithmetic logic unit. A Harvard memory architecture is used to separate instruction and data memory. The memory load/store instructions are 2-cycle operations with one delay slot that can be filled.

The RISC instruction set of the Carmel processor contains several non-RISC instructions to support the execution of FCP. The `deref(Rd2,F,Rs,Rd1,Tag)` instruction is 11 words long, and is used to implement pointer dereferencing. Starting from the address in Rs the dereferenced value is stored in Rd2 and register

Rd1 contains the last reference value. The Tag field is used for subsequent 10-way branching on the dereferenced data type and F is a special-purpose flag. The following four instructions are defined for the manipulation of tags. Instruction `InsTag(Rs, Tag, Rd)` is used to set the destination register Rd with the value part of Rs and the tag value Tag. Instructions `IfTag(Rs, Tag, S2)` and `IfNotTag(Rs, Tag, S2)` are used to branch depending on whether the tag part of Rs is equal (or not) to Tag. The `BRonTag(Rs)` is a ten-way branch on the tag part of Rs. In addition, the `SetTS(C)` instruction sets the *time-slice* register to C and the `IfTS(Y)` instruction performs the *decrement-and-branch-if-zero* operation.

In this dissertation we will compare our work to that of Ginosar and Harsat in several places. First, in Chapter 5, we discuss the design approach for a special-purpose processor architecture given an existing general-purpose implementation. We claim that the analysis must be independent of the host machine characteristics and should be performed at the higher algorithmic level. In [Gino87], low-level machine dependent measures are made to determine time consuming events in the implementation. Using operating system profiling tools, the measures are used to direct the design process.

Also in Chapter 5, our work differs from that of Harsat and Ginosar in the way performance evaluation is performed. We define a system workload that consists of large applications currently in use. Our measurements and conclusions are confined to this workload. Our approach to performance analysis is based on analytic performance models which are also applicable to different workload characteristics. In [Hars88] the performance of the *append* program is reported and it is hard to precisely evaluate the performance of Carmel when it runs large applications.

Special-Purpose Prolog Processors

Several special-purpose processor architectures for the execution of Prolog have been proposed. The Programmable Logic Machine (PLM) [Dobr87], the High-Speed Prolog Machine (HPM) [Naka85] and the Integrated Prolog Processor (IPP) [Abe87] are microprogrammed machines that emulate the Warren Abstract Machine instruction set [Warr83]. In contrast, Pegasus [Seo87] is a VLSI RISC processor for the execution of Prolog.

The PLM is a processor architecture for the execution of Prolog programs within the heterogeneous MIMD multiprocessor system called Aquarius [Dobr85].

The address space of PLM is divided into two areas: the Code space and Data space. The *Prolog Engine* consists of a wide horizontal microcoded control unit and an execution unit that consists of a register file and an ALU with three pairs of dedicated busses. The instructions executed are modified instructions of the Warren Abstract Machine. The specialized instruction set together with architectural support for tag manipulation and the use of a specialized cache enables the PLM to “greatly improve performance” of compiled Prolog execution compared to general-purpose processor implementations [Dobr85]. The HPM and IPP machines architectures follow the same design approach as the PLM.

In contrast to the previously described high-level machine architectures for Prolog, the Pegasus processor is a VLSI RISC processor that includes one interesting feature that deserves attention. The processor register file consists of two parts: the main set of 17 registers and a set of 17 *shadow registers*. Since Prolog creates choice-points while traversing the AND/OR tree, this requires that choice-point registers be saved in a special stack area in memory, much like a context switch. Upon goal failure and backtracking, the saved choice-point registers are restored, that is, moved from memory to the register file.

In Pegasus, instead of moving the choice-point registers to memory, they are written into the shadow register area within a single processor cycle. While the processor continues to execute instructions that do not access memory, the shadow registers are moved to memory. If another choice-point is created before the shadow registers are saved in memory, a *forced shadow write* stage is entered which suspends processor execution until the write phase is complete. A similar situation occurs upon backtracking, where the processor moves the shadow registers into the main set of registers. If further backtracking is required while the processor is prefetching the context into shadow registers, the processor execution suspends until the prefetching is completed. The operations of the shadow registers are dynamically controlled using control bits stored in the processor instructions by the compiler.

Preliminary performance analysis using the *quicksort* program indicate that the performance improvement due to the shadow registers is 18%. Almost one half of the total processor register file size are shadow registers. The register file itself is 20% of the total processor chip area. Since the choice-point size varies with procedure invocation, it is not clear what is the utilization of the shadow registers and whether it is justified to dedicate the amount of processor resources.

A detailed analysis of the Prolog memory reference behavior is described by

Tick in [Tick87]. The minimum choice-point size is 7 and the mean value was measured to be just over 11. The use of a choice-point buffer is described with a hit ratio that levels off at 84% for a buffer size of 12 words. Considering these results, a more cost-effective use of the shadow registers in Pegasus may have been proposed.

3.2 Parallel Inference Machines

Research into parallel inference machines (PIM) was set as the most important objective of the Japanese Fifth Generation Computer System (FGCS) project in the field of computer architecture [Fuch86], [Goto87], [Mura85b]. The programming environment of the inference machines is targeted for the execution of logic programming languages. Initially, the languages selected for parallel implementation were Prolog and Concurrent Prolog [Shap83]. More recently, the committed-choice logic programming language GHC was chosen as the basis for the kernel language KL1 of PIM [Ueda86]. As described in Chapter 2, Flat GHC as well as FCP can be used to model explicit inter-goal communication and synchronization and have been proposed as parallel programming languages for system development [Shap84]. Therefore, FGHC is considered as the implementation language of the Parallel Inference Machine Operating System, PIMOS.

The performance objective for the PIM project was to execute 50-100 K Logical Inferences Per Second (LIPS) on a single processor and 2-5 MLIPS per system of 100 processors, executing the PIMOS operating system. We will refer to the meaning of LIPS and the set objectives later in this thesis. We now review the following special-purpose architectures defined within the parallel inference machine projects:

- PIM based on dataflow machine, PIM-D.
- Reduction based PIM machine, PIM-R.
- Kabu-Wake parallel execution method.
- Personal Sequential Inference machine, PSI and Multi-PSI.
- Parallel Inference Engine, PIE.

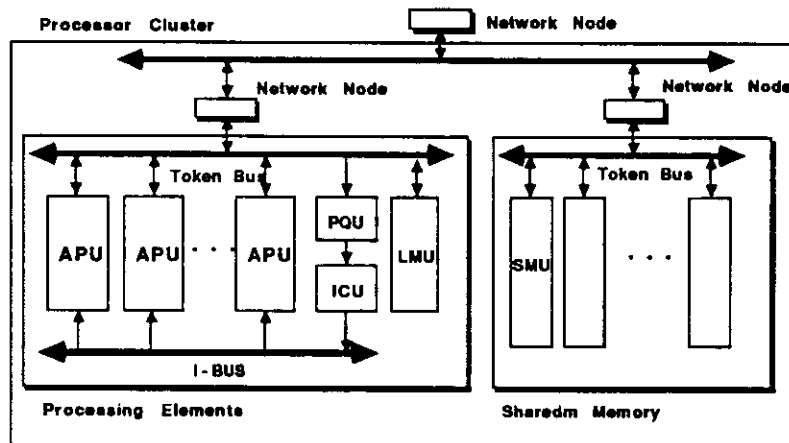


Figure 3.1: The PIM-D Machine

Dataflow Based Parallel Inference Machine, PIM-D

The dataflow based parallel inference machine, PIM-D, from ICOT is described in [Ito85], [Kish85] and [Ito86]. Concurrent Prolog programs are compiled to a dataflow graph with nodes corresponding to primitive operations interpreted by the machine. The parallelism is low-level and inherent in the dataflow graph and not controlled by the user. As in *conventional* dataflow execution models, only those operators that have all operands available are executed sending results on the output arcs.

The PIM-D architecture is shown in Figure 3.1. A single processing element contains one instruction control unit (ICU), one packet queue unit (PQU) and several atomic processing units (APU). PQU receives result packets from the token-bus and passes them to ICU where it is determined which units of computation in the dataflow graph have all operands available. For this purpose, ICU uses hardware hash tables. The ready instruction packets are sent out on the instruction bus to the APU for execution creating more result-packets.

PIM-D is hierarchically structured. More processing elements are connected to each other and to structured memory modules using the token bus, thus forming a cluster. Moreover, clusters are also connected via a network node and the token-bus.

In the software simulations of the PIM-D architecture, it is assumed that the PEs are connected in a 2-dimensional mesh network. The network connecting

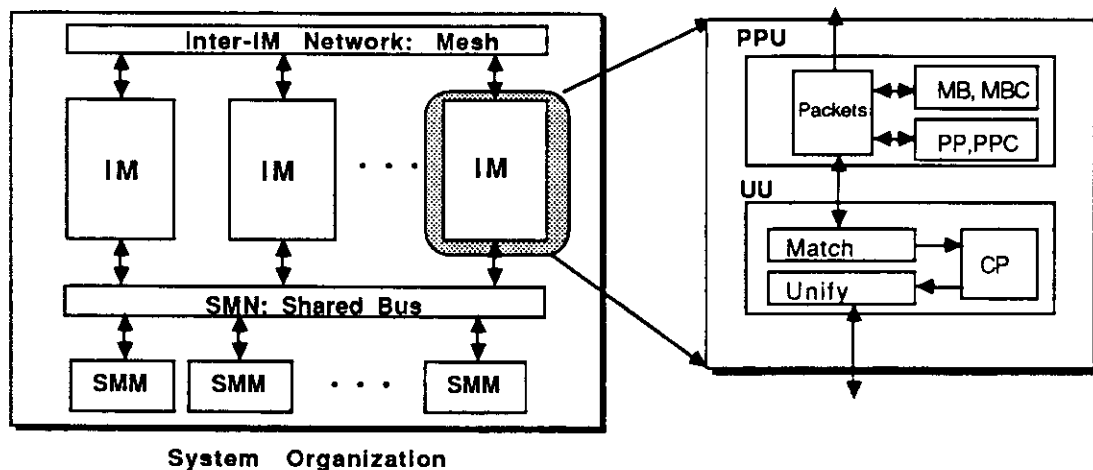


Figure 3.2: The PIM-R Machine

the PEs and the shared memory modules SMs is a multistage switching network and the SMs are also connected via an inter-SM mesh network. Preliminary evaluation using simple applications shows that speedup levels off at 16, when 64 processing elements are used.

However, it is not specified what is the overhead of the parallel model compared to the sequential implementation. In other words, the single processing unit performance should be compared to other single processor implementations.

Reduction Based Parallel Inference Machine: PIM-R

Parallel goal reduction is the basis for the execution model of the parallel inference machine PIM-R. It is used for the AND-Parallel execution of Concurrent Prolog and the OR-parallel execution of Prolog [Mura85a]. In the distributed execution environment of PIM-R, conjunctive goals are executed in parallel on different processors, using shared logical variables as communication channels. The OR-parallelism in the guards of a Concurrent Prolog clause is not distributed but executed on the same processor as the parent goal.

The organization of PIM-R shown in Figure 3.2 is described in [Onai85a], [Onai85b] and [Onai85c]. It consists of inference modules (IM) and structure memory modules (SMM) interconnected using two networks. The inter-IM net-

work is a mesh and the IM-SMM network is a shared bus. The structure memory modules implement a distributed shared memory system. One IM is located at each mesh node and consists of a process pool unit (PPU) and a unification unit (UU). The unification unit contains a copy of the entire program in the clause pool memory. In addition, UU consists of a clause-matching unit and a unifier. UU receives a process from PPU and the matching unit tries to find a matching clause by searching the clause pool. Matching clauses are sent to the unification unit for reduction. Reduced clauses are sent back to PPU for creating new processes stored in PPU. Process suspension is implemented using a message board for storing shared logical variables as channels. The message board is controlled by the message board controller.

When PPU receives reduced clauses from UU, conjunctive processes are distributed via the inter-IM network to those IM with the shortest input buffer length. The network nodes dynamically control the distribution of processes. For this purpose, special node controllers of the Inmos Transputer type are proposed.

In [Onai85a] the speedup of executing the *quicksort* program in Concurrent Prolog on 16 inference modules is just over 2. The following are some of the reasons for performance degradation: frequent goal suspensions and activations, frequent control messages distributed over the network and poor load distribution and balancing. Moreover, the same comment made for the PIM-D machine can be made here as well. By specifying only the speedup values, it is not clear how the single inference module compares to other single-processor implementations.

The Kabu-Wake Method

In [Sohm85] and [Kumo86] a parallel inference method for the OR-parallel execution of Prolog is proposed called *Kabu-Wake*. The name refers (in Japanese) to the method of splitting a tree at a node and then replanting it at a different location. If we recall the AND/OR tree computation model for logic programs described in Chapter 2, the Prolog sequential execution performs the depth-first search using an efficient stack control mechanism. Alternative paths in the tree are stored on the stack, and are considered only if the current path of execution fails.

It is the efficient sequential implementation that has motivated the Kabu-Wake method. Each busy processor in a distributed Kabu-Wake machine exe-

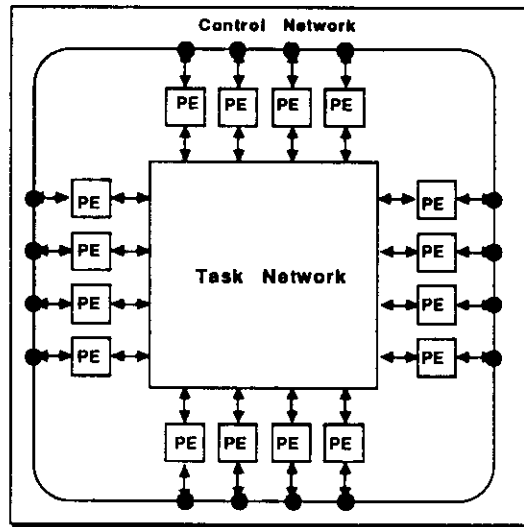


Figure 3.3: The Parallel Inference Machine for Kabu-Wake Execution

cutes sequential Prolog. An idle processor sends a message to a busy processor requesting work. In response, the busy processor services the request by first leaving the current busy execution mode and backtracking to the choice-point closest to the root of the AND/OR tree. The tree is then split into half, packaged and sent to the idle processor to execute. The execution model at each processor continues to be sequential, as if executing sequential Prolog.

The following feature of the Kabu-Wake method should be clear. The parallelism that is exploited between processors is strictly OR-parallelism. The sequential Prolog execution model on a single processor would eventually reach the split subtree if all the subtrees to the left of it fail. Since Prolog unbinds variables upon failure and backtracking, so the Kabu-Wake method must unbind all variables that belong to the left of the subtree prior to sending. This may be a source of performance degradation.

In Figure 3.3 we show the parallel inference machine proposed for the execution of the Kabu-Wake method. It consists of 16 general-purpose processors with local memory. The processors are connected via two different networks called the *data network* and *control network*. The data network is a multi-stage switching network used for transferring parts of the subtree from one processor to another. The control network is a ring bus used for reporting the status of each processor

and for requesting subtrees from busy nodes.

Preliminary performance evaluation of the *parse* program shows that the method is sensitive to the granularity of parallelism. The main overhead is in splitting the subtree and preparing it for sending to a remote processor. The speedup depends on how effectively the program application is split and whether it is split close to the root or not. Running on 16 processors a specific parse sentence showed a speed-up of 10.8 relative to execution on a single processor. However, introducing the Kabu-Wake method on a single node results in an overhead of sequential execution due to the special-treatment of the logical variable.

PSI-II and Multi PSI Architecture

As a result of research of the previously described inference machines, it was clear that a simpler language and execution model were required to reap the benefit of parallel execution in a distributed environment. For this purpose, the *core* of a kernel language KL1 was defined based on Flat GHC (FGHC). A distributed implementation of FGHC on a network of Personal Sequential Inference machines (PSI) called Multi-PSI is proposed. The first version of the uniprocessor is called PSI-I and the subsequent improved version PSI-II.

The PSI-I was designed as a special-purpose high-level language processor for the execution of logic programming languages. It is a microprogrammed architecture with 16 Kwords of 64-bit writable control store used for the execution of a logic programming interpreter. It has special-purpose hardware support for the manipulation of data tags and for the manipulation of multiple-stacks. It is implemented in TTL MSI technology and has a processor cycle of 200nsec.

The PSI-II [Naka87], is an improved version of the PSI-I, both in terms of performance and cost. Compared to PSI-I, the PSI-II was designed with three objectives in mind: to reduce the hardware size, to improve the processor performance and to accommodate extensions for a Multi-PSI processor implementation. The hardware size was reduced by using a higher scale of integration and by building custom LSI modules. Using gate-array CMOS technology the size of PSI-II is reduced to one quarter of the size of PSI-I. To increase system performance, the microcoded interpreter was replaced by a microcoded special-purpose instruction set that is suitable for compilation and compiler optimization techniques. The performance improvement resulted in program speedups of 3 to 10. Finally, the PSI-II is designed with multiprocessor extensions in mind. The

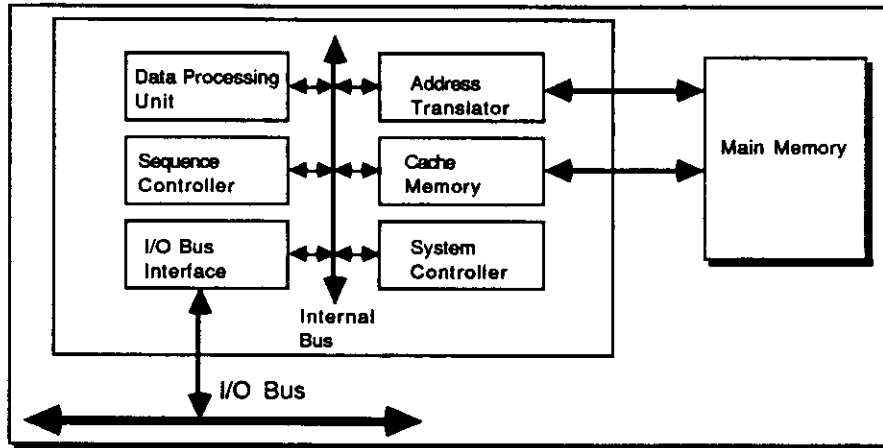


Figure 3.4: PSI-II Machine Organization

extensions include a larger instruction format to accommodate future special-purpose instructions used in a parallel implementation; a larger memory address space with reserved address areas for use in the multiprocessor machine and more. The system organization of the PSI-II is shown in Figure 3.4. It has a machine cycle of 166.7 nsec.

The Multi-PSI [Kish86] is a multiprocessor machine that connects up to 64 PSI-II processors in a 2-dimensional mesh network, as shown in Figure 3.5. Each processor in the network has 16 Mwords of 40bits local memory and communicates with other processors via a network communication controller.

The distributed implementation of Flat GHC on the Multi-PSI is described in [Ichi87]. The Multi-PSI does not assume a global address space. Therefore, there are no explicit *remote references* to locations at remote processors. Rather, an indirect addressing mechanism is maintained at each processor. The indirect addressing is implemented using a variable address management table at each processor. The motivation for this approach is found in the localized rather than distributed garbage collection algorithm that can be applied. When remote references are used, garbage collection at one processor must follow the chain of remote references and induce a distributed garbage collection algorithm. This may create a bottleneck. Using the indirect table mechanism allows local garbage collection. As data structures are relocated locally, their indirect address pointers are updated in the variable management table. There is no need to access remote processors. However, an analysis of the tradeoffs involved is necessary.

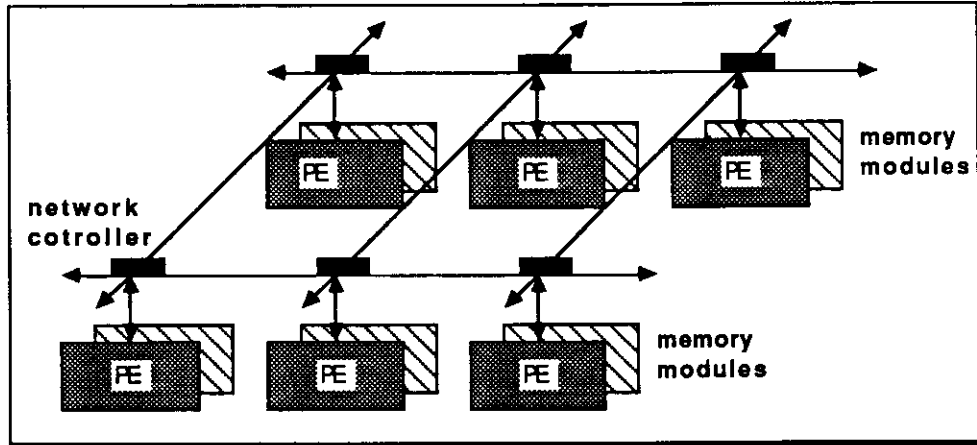


Figure 3.5: Multi-PSI Organization

Conceptually, each processor in the Multi-PSI machine consists of a active queue of reducible goals, an input and output communication channels and a variable management table. Goal reduction at one processing element may result in the “*throwing*” of goals to remote processors to reduce. The goal migration is controlled by user defined *pragmas*. As of yet, a detailed performance evaluation of the distributed implementation of FGHC on the Multi-PSI is not available.

Parallel Inference Engine: PIE

A Parallel Inference Engine (PIE) for the execution of AND-parallel languages is proposed in [Koik86] [Tosh87]. It implements a subset of the language GHC without guards, called FLENG. The language models goal suspension and activation in the same way as GHC and can thus be used to implement GHC as a higher level language.

The PIE system consists of multiple *inference units* (IU) connected using two different networks. One is an *omega* packet switching network used for distributing goals between the inference units, (DN), and the other is a shared memory network (SMN) for managing shared variables and structures as well as activating suspended goals. The type of network used for SMN “has not been decided yet” [Tosh87]. The organization of PIE and the inference units is shown in Figure 3.6.

Each IU contains a *unification processor* (UP), program definition memory

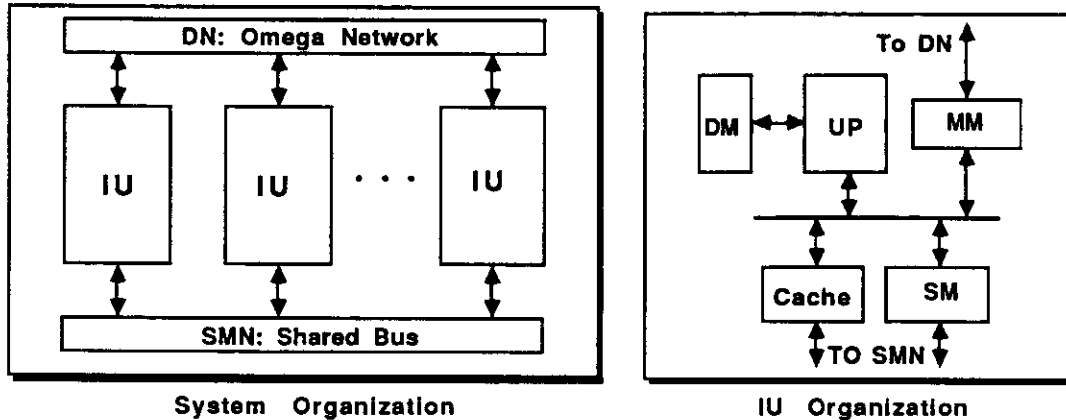


Figure 3.6: The Parallel Inference Engine for FLENG

(DM), a memory module (MM) connected to the distribution network used for storing goals, a distributed shared memory (SM) and a shared memory cache (C). The definition memory in each IU contains a copy of the complete FLENG program.

The parallel execution model of PIE described in [Tosh87] specifies that goals spawned as a result of a goal reduction are “distributed into different MMs”. To reduce the use of the SMN, goal argument structures are copied and distributed as well. Goals are distributed via DN according to the *load* in each IU, where the load is determined according to the number of active goals. When UP spawns a new goal, it is sent to the IU with the least number of active goals.

Goal suspension is implemented by storing a goal pointer in the suspension variable and changing the status of the goal from active to suspended. When the variable receives a value, a message is sent via SMN to the IUs with the suspended goals. In response, the suspended goal is reactivated. If a message is received for a goal that is being executed, the UP aborts execution and restarts with the newly received value stored in the cache.

A preliminary evaluation of PIE is described in [Tosh87] using simulations and executing four simple programs: *quicksort*, *naive reverse*, *primes* and *permute*. It is noticed that the frequency of process suspension and activation degrades system performance. Moreover, the following two issues remain a problem in the architecture. First, the overhead of accessing remote shared memories and the existence of a suitable network that connects the distributed memory modules. For systems of more than 64 processors, a cluster architecture is proposed. The

second issue is the problem of load-balancing and distribution of reducible goals to remote processing units.

3.3 Shared Memory Inference Machines

Parlog on the Sequent Balance Multiprocessor

The implementation of Parlog on the Sequent Balance multiprocessor is described by Crammond in [Cram88]. The Sequent Balance is a shared-memory general-purpose multiprocessor composed of NS32032 processors connected on a shared bus. This is a departure from the previously described execution models that used distributed rather than shared memory architectures. Each dual-processor board shares a 8Kbyte write-through cache that follows a cache coherency protocol. In addition, a separate SLIC Bus is used to connect the System Link and Interrupt Controllers (SLIC) that are used for low level control messages and interrupts.

The parallel implementation of Parlog on the Sequent Balance relies on the existing operating system *Dynix* which supports parallel programming. At the operating system level, a separate Parlog abstract machine runs on each node of the multiprocessor. Each abstract machine has a separate *run-queue* used for scheduling goals locally. In case one of the queues becomes empty, a *search for work* algorithm is used to find a non-empty queue from which to *steal* executable goals. Preliminary performance evaluation of six small programs show in most cases “very good” speedup of 12 to 15, when executed on 20 processors.

Aurora OR-Parallel Prolog System

Aurora is a prototype implementation of OR-parallel Prolog on a shared memory, general-purpose, multiprocessor machine [Lusk88]. The objective of Aurora is to effectively implement full Prolog using the implicit OR-parallelism in a transparent way. Since Prolog is efficiently implemented on a general-purpose machine by emulating the sequential abstract machine proposed by Warren [Warr83], parallel execution models of Prolog are proposed as extensions to the emulator for the purpose of exploiting implicit parallelism whereas preserving as much of the sequential efficiency as possible.

In Aurora, the AND/OR tree that is sequentially searched in Prolog (depth-

first), is searched in parallel by independent *workers*. A worker may be a physical processor or a process in a multiprocessing environment. Each worker executes part of the AND/OR tree as if it is a separate Prolog session. When a worker is out of work, the *scheduler* is responsible for matching the worker with available work searching the tree.

The main problem in implementing OR-parallel search tree execution is the management of shared variable bindings that belong to different execution paths. In Aurora, the SRI model is used [Warr87]. In this model, shared logical variables have no special treatment if they are allocated on a deterministic path in the tree. However, with the creation of the first choice-point, the allocated variable may potentially have more than one value. To manage different instances of the same variable in the SRI model, a worker maintains a *private binding array* for storing conditional bindings. A shared variable that is conditionally bound stores an offset into the binding array where the value is found.

The maintenance of binding arrays results in significant overhead mainly when a worker needs to look for more work, thus switching to another part of the search tree. As a worker moves up or down the tree looking for work, so it must update its binding array.

The preliminary performance evaluation of the Aurora system using simple applications shows encouraging results and speedups of up to 14, on 16 processors. The modifications made to the sequential Prolog emulator result in a 25% overhead when Aurora executes on a single processor.

Comparison of KL1 and Aurora Execution on the Sequent

The analysis of AND-parallel committed choice execution of KL1 (FGHC) and the execution of the OR-parallel non-committed choice system Aurora is described in [Tick88]. Both languages are implemented on the Balance Sequent multiprocessor machine. The empirical evaluation is performed using non-trivial benchmark programs optimized for performance in both languages. Various memory hierarchy organizations are considered including coherent cache policy alternatives.

The following two observations made in [Tick88] deserve attention. First, Aurora (non-committed choice language) has “better” memory performance characteristics than FGHC (committed-choice language). The reason is that Aurora

makes use of the more efficient stack-based storage model of sequential Prolog compared to the heap-based storage model of FGHC. This results in reduced memory reference locality which in turn reflects on the memory performance of data caching.

The second observation is that committed-choice execution of FGHC performs better than Aurora for single-solution problems, while Aurora performs better in applications where multiple solutions are required. In other words, since single solution problems tend to have more fine grain concurrency, and since the overhead of creating new tasks is greater in OR-parallel systems than in AND-parallel systems, this explains why single solution problems execute more efficiently in KL1 compared to Aurora.

The motivation for performance analysis described in [Tick88] as well as the analysis approach are further discussed and compared to the analysis approach in this thesis, later in Chapter 5.

The PEPSys Machine Architecture

The Parallel ECRC Prolog System (PEPSys) is a research project to design and evaluate a multiprocessor system for the execution of large scale parallel Prolog applications [Baro88a]. In contrast to the SRI model used in the Aurora system which makes use of only OR-parallelism, PEPSys defines both OR-parallel and *restricted* AND-parallel execution which is specified by the user using procedure declarations. Restricted AND-parallelism allows the concurrent execution of only those conjunctive goals that do not share variables or have fully instantiated variables so that no conflict can occur. The result is a more complex implementation with an objective to exploit more of the potential concurrency inherent to the AND/OR tree search. Even though the preliminary analysis of program execution on a shared memory multiprocessor with 8 processing elements shows encouraging speedup, a meaningful comparison of the parallel Prolog execution models as well as the analysis of tradeoffs between concurrency and overhead of implementation has not yet been made.

The architecture of the PEPSys machine is described as a system of distributed clusters, with each cluster being a shared memory multiprocessor. In addition to a set of processing elements that share a common bus inside a cluster, a special-purpose Cluster Processor (CP) performs inter-cluster communication via message passing. Each cluster has a local *workpool* of concurrent OR and

AND processes. While the workpool is non-empty, the processing elements execute processes stored in the shared memory. When one of the processes becomes idle and the workpool is empty, a message is sent to CP reporting its idle status. CP maintains a list of cluster workload distribution, and sends a message to a selected remote cluster requesting work. The cluster receiving the message can either ignore the message or alternatively *search for work* and respond by sending work to the cluster with the idle processing element.

Software simulation and evaluation of the cluster architecture executing the *8 queens* problem is described in [Baro88b]. Increasing the number of processing elements in a cluster results in performance improvements but increasing the number of clusters for the same number of processing elements degrades performance considerably. The following reasons are cited. First, the load balancing scheme that takes work from a remote cluster does not result in effective load sharing unless care is taken not to share fine grain computations. Second, the cluster processor needs to perform faster than the processing elements in the cluster to avoid long idle times. And finally, the *8 queens* problem is hardly representative of a large scale application and thus the evaluation of the PEP-Sys cluster architecture may not be appropriate using a program that does not exhibit enough parallelism.

The Aquarius Multiprocessor Machine

The Aquarius multiprocessor machine consists of Parallel Prolog Processors (PPP) that share a multi-module memory system. The PPP is an extension of the PLM processor for the execution of Prolog, described earlier. In the considered machine organization, each PPP can access 16 memory modules via a cross-bar switch and a special-purpose synchronization memory via a shared bus. A cache is used between each PPP and each main memory module in addition to a cache placed between each PPP and the synchronization memory. The organization of the Aquarius multiprocessor shown in Figure 3.7 is described in [Fagi87a].

The multiprocessor system of PPPs executes Prolog in both OR-parallel and Restricted AND-parallel mode as described in [Fagi87b]. In addition, *intelligent backtracking* algorithms are used to improve performance. The performance results presented in [Fagi87a] are obtained from simulating an ideal multiprocessor system with 4 and 8 processors executing a set of benchmark programs previously used for the evaluation of Prolog systems. The author recognizes very poor performance improvements due to parallel processing using the PPP model, and is

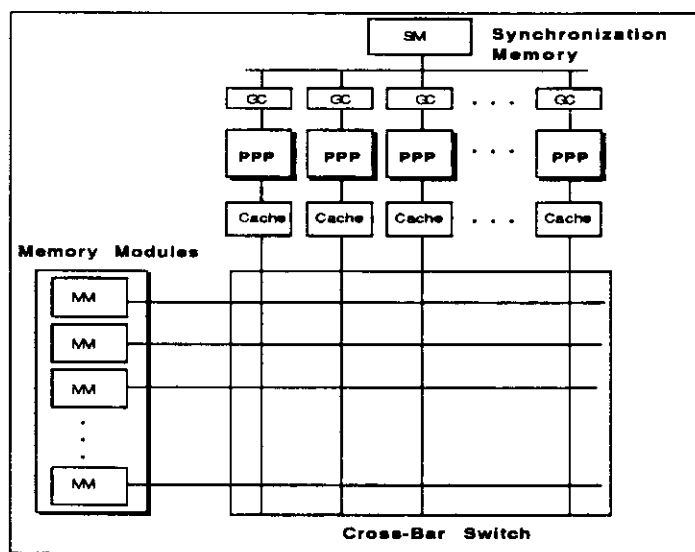


Figure 3.7: The Aquarius Multiprocessor Machine

doubtful that the benefit of concurrency can outweigh the cost of multiprocessing in a parallel Prolog model like PPP. The reasons cited are the overhead of process creation and termination, scheduling, synchronization and in general the cost associated with multiprocessing. However, the results and conclusions are based on applications that are not representative of a real system workload. Better results may be achieved using programs that exhibit more inherent parallelism.

3.4 Chapter Summary

We have reviewed special-purpose architectures for logic programming languages that are related to our work. Therefore, this chapter is not intended as a complete review of the field but to identify some of the common research directions and problems. Common to all the previously described architectures and logic programming execution models is the lack of a systematic approach to performance evaluation. Preliminary evaluations using simple programs that are not representative of a system workload are repeatedly performed. The work of Tick [Tick87] and Fagin [Fagi87b] are encouraging attempts to rectify this trend. In this thesis, we contribute by suggesting a methodology for performance evaluation based on analytic performance models. The empirical approach to evaluation

is based on a set of programs representative of a specific system workload.

Except for the Aurora system, none of the previously described parallel implementations of logic programming languages, specify the overhead of the parallel model compared to sequential execution. Without this information, the speedup numbers based on simple applications are less meaningful.

The main problems that are cited in the implementations of parallel logic programming languages is the overhead of goal suspension, activation, context switching, task creation, and in general the complexity of the parallel model. Other issues include load balancing, overhead of inter-processor communication etc. In this thesis, we address these issues and suggest ways to reduce the overhead of goal management in FCP, which includes goal spawning, goal termination, suspension activation and goal (context) switching. Goal management is one aspect of the special-purpose FCP processor proposed.

CHAPTER 4

Implementations of FCP

In this chapter we describe the following previously proposed implementations of FCP on existing general-purpose architectures:

1. Sequential Interpreter
2. Sequential Abstract Machine
3. Distributed Interpreter
4. Distributed Abstract Machine

The first two represent sequential control models, written in Pascal and C respectively, that execute on general-purpose uniprocessors. The latter two represent parallel control models for the execution of FCP in a distributed environment. The distributed interpreter is written in Occam and its execution on a system of Transputers is simulated. The distributed abstract machine is emulated in C and implemented on the Hypercube. Both distributed implementations are defined as extensions of the sequential execution models. For each implementation we describe the run time environment and the control algorithms. Where available, we also discuss the preliminary performance evaluations and reported implementation problems.

4.1 FCP Sequential Interpreter Implementation

An interpreter-based implementation of FCP is described in [Mier84]. It is composed of two levels. The *lower level* consists of an interpreter of the language kernel, written in Pascal, and the *higher level* consists of a boot, tokenizer, parser and other programs written in FCP that define the language support and user interface. In the following section we briefly discuss only the lower level since it directly led to the development of the compiler-based implementation described in Section 4.2 .

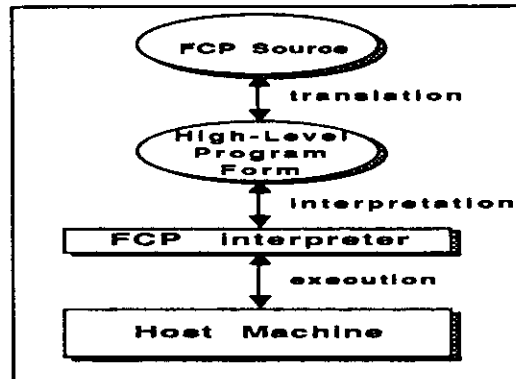


Figure 4.1: FCP Program Interpretation

Program Representation

Figure 4.1 symbolically represents the interpreter level of program execution. FCP source programs are translated to a high-level representation and then symbolically interpreted on the host machine. The program is represented as a list of procedures where each procedure is a list of clauses. A clause is represented as a structure with three arguments denoting the clause-head, clause guard and clause-body.

Run Time Environment

The run-time environment consists of a heap, a Trail Stack (TS), a Suspension Variable Table (SVT), a Functor Table (FT) and a String Table (ST), as depicted in Figure 4.2. The heap is implemented as an array of tagged words. It is used for storing all program terms, that is, structures, lists, constants, references and variables, and also for storing programs and the program control structures. Memory for program execution is allocated during a clause-try from the top of the heap using the HP pointer. The HB pointer marks the top of the heap prior to the clause-try. It is used for garbage collecting the top of the heap during shallow backtracking. That is, when a clause-try fails, the top of the heap is reclaimed by assigning HB to HP. When the end of the heap is reached, a stop-and-copy garbage collection algorithm is used to reclaim unused data structures only in the heap. An FCP program is stored as a program data structure on the

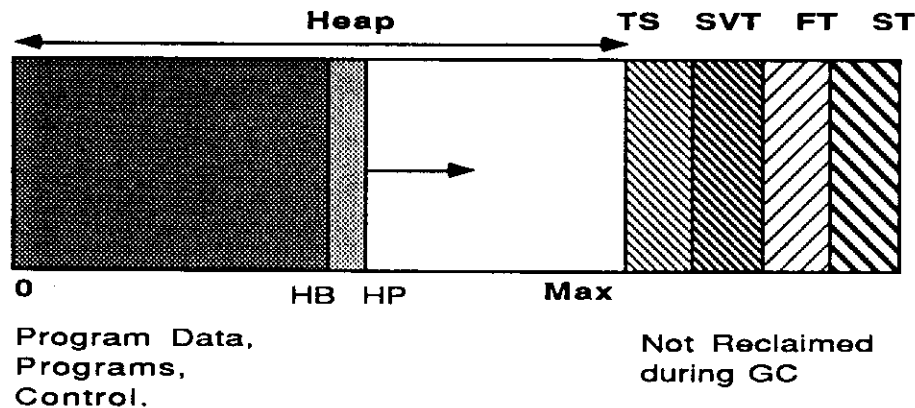


Figure 4.2: FCP Interpreter Run-Time Environment

Heap.

Two types of program control structures are also allocated on the heap: goal records and suspension records. A goal record contains a pointer to a goal, a pointer to the stored program, and a pointer to the next goal record. The set of active goals ready for reduction is managed as a queue of goal records. The pointers QF and QB denote the front and end of the active goal queue. In addition, a goal free list pointer GFL is used to link discarded goal records after the goal is reduced.

Suspension records linked in *suspension lists* are used to implement the goal suspension and activation mechanisms. The suspension list starts in the variable on which the goal suspends on. The suspension records are collected onto the suspension free list SFL upon goal activation.

TS is used to store changes to the program environment during a clause-try. Each entry in the stack consists of a variable address and previous value. The variable value is either a *null* value if the variable was previously uninstantiated, or a pointer to a suspension record. If the outcome of the clause-try is *fail* or *suspend*, the entries in TS are used to restore the program environment to the state prior to the clause-try. If the outcome is *commit*, then TS is used to access suspension record pointers and activate suspended goals.

SVT is used during a clause-try to store addresses of variables on which a goal may suspend. In case of a clause-try failure or commit, the table is reset.

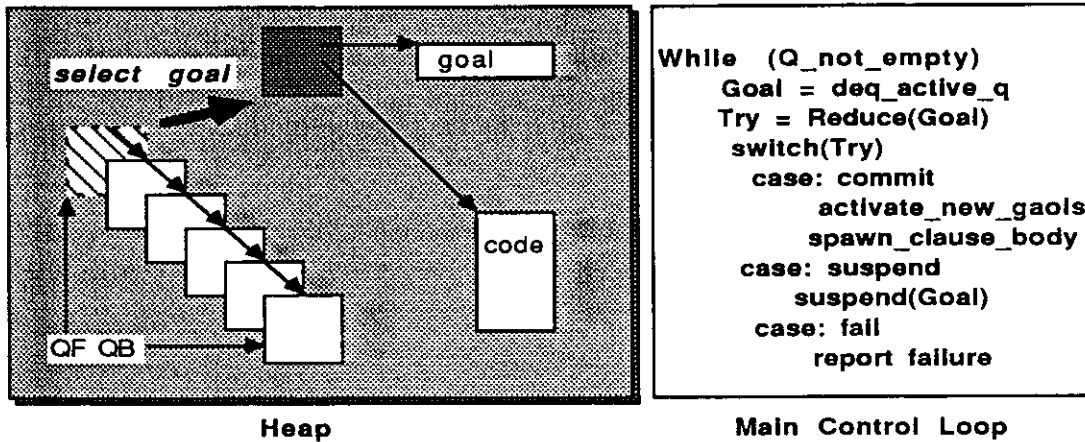


Figure 4.3: Goal Record Selection

Only in case of a goal suspension are the entries in the table used to implement the goal suspension mechanism.

FT is used as a hash table for the access of functors by name and arity. The stored values represent entries into the ST, which stores all program strings.

Interpreter Control

The active set of goals in the FCP interpreter implementation is represented using a queue denoted by the QF and QB pointers, as shown in Figure 4.3. Selecting the next goal for reduction consists of accessing the front of the queue, removing the goal and updating the queue pointer.

Goal reduction consists of a series of clause-tries performed in textual order, as shown in Figure 4.4. A single clause-try is implemented as a function call that returns either *true* (if it succeeds) or *false* (if it fails or suspends). For each pair of arguments in the clause-try, a *unify(goal_arg, clause_arg)* function call is made that also evaluates to *true* or *false*. Goal-head unification succeeds if all the individual argument unifications evaluate to *true*, otherwise it fails.

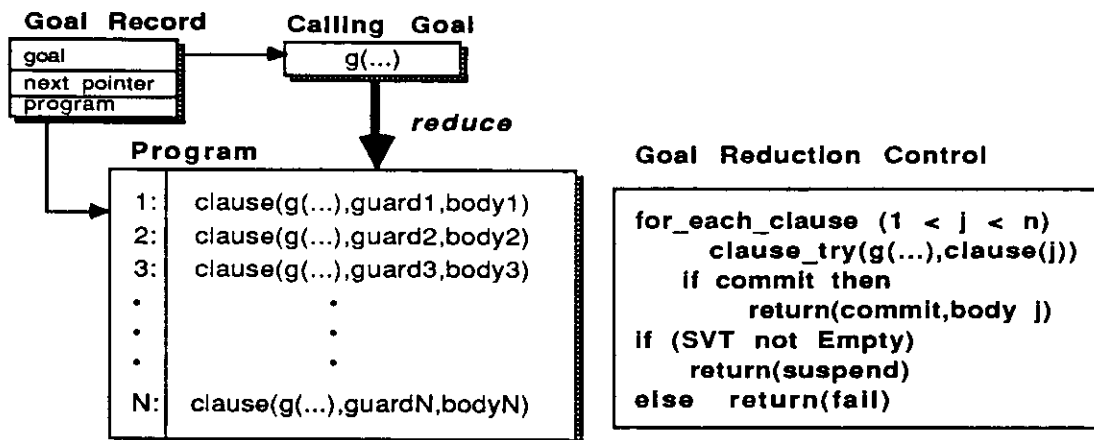


Figure 4.4: Clause-Tries in Textual Order

4.1.1 FCP Interpreter Characteristics

The above described interpreter represents the first implementation of FCP on a general-purpose machine. The data structures and control mechanisms defined in the interpreter implementation, as well as the top level user support were fundamental for the design of the compiler implementation discussed in Section 4.2. The main motivation to develop a compiler-based architecture was the inadequate performance of the FCP interpreter implementation and the potential for increased performance a compiler may provide. A detailed performance analysis of the FCP interpreter was not performed; the only performance related result available quoted an execution rate of 386 Logical Inferences Per Second (LIPS) for the interpreter execution of the *naive reverse(100)* program running on the VAX 11/750 machine. By dividing the FCP interpreter level into a sequence of primitive operations, a wide range of compiler optimization techniques become available. These are discussed in following section.

4.2 FCP Sequential Abstract Machine Implementation

The compiler-based sequential abstract machine for the execution of FCP [Hour86], resulted directly from the FCP interpreter discussed in the previous section. To understand the continued process towards compilation, let us recall the level at which FCP programs were interpreted. That is, FCP programs were translated into a high-level clausal representation with the interpretation algo-

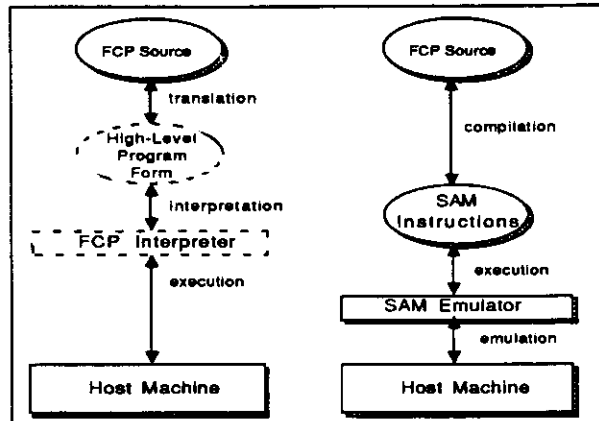


Figure 4.5: Interpreter and Compiler Oriented Machine Implementation

rithm executing at the clause-try level. Therefore, clause-head unification was interpreted and not divided into a set of primitive operations. It is in this direction that the compiler-oriented implementation extends the translation process to a level closer to the general-purpose host machine. In Figure 4.5 we show the continued process of FCP translation by placing the compiler target machine level below the interpretation level and closer to the physical machine level.

Therefore, the FCP sequential abstract machine is based on the following two features:

- The interpreter run-time architecture and control form the basis for the compiler-oriented implementation.
- Program representation, clause selection and clause-level interpretation are replaced by abstract machine instructions similar to the Prolog abstract machine instructions defined by Warren [Warr83].

We now describe the abstract machine run-time environment followed by the abstract machine instructions and the compiled program representation.

Run-Time Environment

The FCP abstract machine run-time environment is shown in Figure 4.6. It is a slightly modified version of the FCP interpreter environment. By allowing

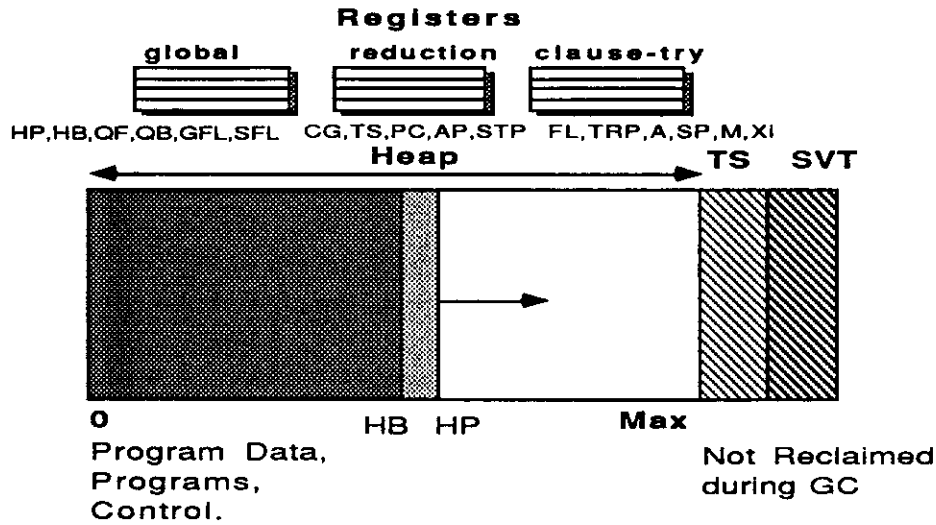


Figure 4.6: FCP Abstract Machine Run-Time Environment

string data structures to be represented on the Heap, FT and ST are eliminated as special-purpose areas. Also, a goal record is redefined to include the goal arguments explicitly, instead of pointing to a goal representation. The most important addition to the run-time environment is the notion of abstract machine registers that denote the abstract machine state and which are used by the abstract machine instructions. The following three functional sets of registers are defined:

- Global Registers: HB, HP, QF, QB, GFL, SFL
- Goal Reduction Registers: CG, TS, PC, AP, STP
- Clause-try Registers: FL, TRP, A, M, SP, X_i

The global registers are used for Heap management in the same way as it was described in the FCP interpreter. During a goal reduction the current goal pointer is stored in CG and the current program counter in PC. To prevent an infinitely recursive goal from depriving other goals of execution time, a time slice is associated with each recursive goal reduction. The time slice value is stored in TS and is decremented at the end of each iteration. If the time slice is exhausted, a goal switch is induced and a new goal is selected for reduction, while the current goal is placed at the end of the active queue. The value of the time slice can be varied. During goal activation, a pointer to the activation list is stored in AP,

and during a clause-try STP denotes the next entry in SVT for storing suspension variable pointers.

During a clause-try FL stores the Failure Label used for selecting the next clause in case of clause-try fail or suspend. The Trail Stack Pointer TRP denotes the top of Trail Stack, A keeps the goal argument pointer index, SP the structure pointer index used during unification and M stores the unification mode of operation. The X_i registers denote a set of temporary registers that can be used by the abstract machine compiler.

4.2.1 Abstract Machine Instructions

The FCP sequential abstract machine instructions are based on the WAM instruction set for Prolog. They are grouped into the following categories:

- Indexing Instructions
- Clause-Head Unification Instructions
- Guard Instructions
- Argument Creation Instructions
- Goal Management Instructions

The main difference between the instruction set of the FCP abstract machine and the WAM instructions is the addition of goal management instructions as well as the modification of unification instructions to handle *read-only* variables. This type of unification is also referred to as *read-only unification*. We now briefly review only the use of the abstract machine instruction groups. A detailed list of the abstract machine instructions can be found in [Hour86].

Indexing

Indexing instructions are used to select a clause during a goal reduction. The `try_me_else(Label)` instruction precedes a clause-try sequence of instructions and specifies the address of the next clause-try. The `Label` value is stored in the FL register during clause-try execution. If the current clause-try fails, the next clause-try denoted by the FL register is selected.

Clause-Head Unification

Two types of abstract instructions are used during goal-head unification: `get` and `unify` instructions. The `get` instructions define the structure of the clause-head argument and are thus used only at the top level of argument unification. Subsequent nestings of argument unifications are encoded using the `unify` instructions. For example, to match a clause-head that has n arguments with a calling goal, the `get` instruction type will be used n times. However, for each argument, the matching of nested structures is performed with the `unify` type of instructions.

It should be pointed out that we refer to the `get` and `unify` instructions as *types* of instructions. That is, they denote a group of instructions that can be tailored for a specific type of argument. For example, the `get` instruction type includes instructions such as: `get_variable(X)`, `get_list(X)` etc. Besides the treatment of *read-only* variables, these instructions are the same as the WAM instructions.

Guard Calls

The `call(Arg,Index)` abstract instruction represents a function call to a pre-defined primitive operation used only in the clause guard. The guard call is identified using `Index`. `Arg` denotes the arguments of the function call.

Argument Creation

After a clause-try commits, the arguments of the body goals are created using two types of instructions: `put` and `unify`. The use of these instructions is similar to the use of `get` and `unify` instructions in the clause-head unification. That is, the `put` instructions are used to form the top level structure of the body goal. Successive `put` instructions correspond to new goal arguments. For example, to create n arguments of a body goal, the `put` instruction type will be used n times. In between the `put` instructions the `unify` type of instructions are used to create nested argument structures.

Goal Management

Five goal management instructions are defined in the FCP abstract machine: `commit`, `spawn(Goal)`, `execute(Goal)`, `halt` and `suspend`. The `commit` instruction

denotes the end of a successful clause-try and consists of activating goals using the suspension pointers trailed in TS. The `spawn(Goal)` instruction allocates and places a goal record onto the active goal queue. Subsequent `put` instructions place the argument pointers into the goal record. The `execute(Goal)` instruction is used to iterate, and execute Goal without selecting a new goal from the active goal queue. The time slice is thus decremented to limit the number of iterations and to force a goal switch if the time slice is exhausted. The `halt` instruction marks the termination of the current goal, collects the goal record and schedules a new goal from the active goal queue. Finally, the `suspend` instruction results in the suspension of the current goal using the variables stored in SVT.

In the following section we describe how FCP programs are compiled to the abstract machine level using the above defined instruction types and specific instructions.

4.2.2 FCP Abstract Machine and WAM

The FCP abstract machine was significantly influenced by the design of the Prolog abstract machine (WAM) [Warr83]. General unification is implemented using the same `get`, `put` and `unify` instructions extended to handle the unification of *read-only* annotated variables [Hour86]. Several important features distinguish the FCP abstract machine from WAM.

- Non-deterministic goal scheduling.
- Goal suspension and data flow activation.
- Heap storage model.

As a concurrent logic programming language intended for parallel processing, the program *resolvent* is modeled as a set of non-deterministically scheduled goals, much like processes in a multiprocessing system. In the FCP abstract machine, a queue of goals is used instead of the stack-based control used in WAM.

To model inter-goal communication and synchronization, FCP and concurrent logic programming languages in general, define *rules of suspension*. A goal *suspends* if there is insufficient data to perform successful goal-head unification whereas the reduction may succeed when the data becomes available. In the FCP abstract machine, goal suspension is implemented using *suspension lists*

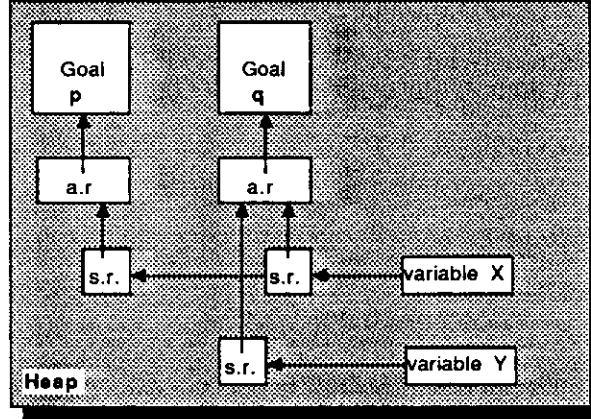


Figure 4.7: Goal Suspension Mechanism

associated with each suspending variable. Data flow activation is implemented by rescheduling the suspended goals when the suspending variables receive data from other active goals. In Figure 4.7, we show goal *p* suspended on variable *X* and goal *q* on variables *X* and *Y*. *Suspension records* are linked into suspension lists if more than one goal is suspended on the same variable. Single *activation records* per suspended goal prevent multiple activation of the same goal.

The third important feature that distinguishes the FCP abstract machine from WAM is the storage model. FCP uses a *heap-based* model as opposed to the stack-based model, to store both program and control structures. The common heap is an essential part of the parallel programming model. It enables the implementation of asynchronous inter-goal communication using dynamically allocated logical variables as a continuous communication *stream*.

4.2.3 Program Representation

In the FCP abstract machine a program clause is stored on the Heap, represented as a sequence of abstract machine instructions. The assembled instructions represent the unfolding of the FCP clause-level interpreter execution, specialized according to the structure of the clause arguments. Since the structure of the clause arguments is known at compile time, it is possible to assemble abstract machine instructions according to the clause arguments' type.

For example, a clause-head with n arguments may be compiled to a sequence

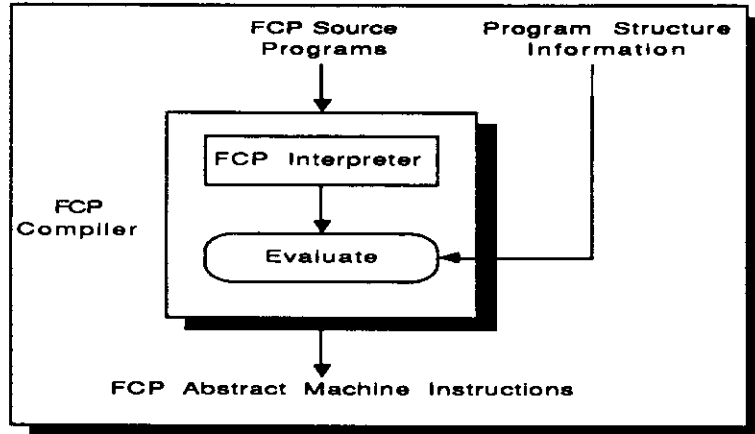


Figure 4.8: FCP Abstract Machine Compiler: Evaluate Interpreter

of n unification instructions rather than having the single clause-try interpreter perform the complete clause-try. Moreover, if one knows that the first two arguments of the clause are a *list* and *constant*, there is no need to call the general unification algorithm. Instead, the specialized instructions `get_list` and `get_constant` represent the same unification call partially evaluated according to the data types known at compile time. In Figure 4.8 we show the continued transformation of the source program using the interpreter algorithm partially evaluated with compile time information about the program structure.

We now discuss the encoding of an FCP procedure followed by the representation of a single clause in the procedure.

Procedure Encoding

In Figure 4.9 we show the encoding of a procedure that consists of n clauses. Each block of abstract machine instructions for a single clause is preceded by the `try_me_else(Label)` instruction. If the last clause-try fails, the goal reduction will suspend if the SVT is empty, otherwise an error message is generated.

Clause Encoding

In FCP, three types of clauses are distinguished, as shown in Figure 4.10, for the general case of a predicate relation p . Let g_i denote a set of predefined

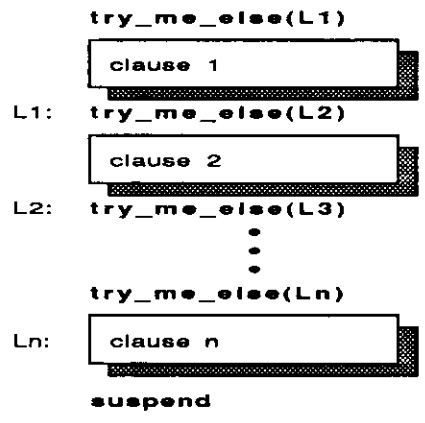


Figure 4.9: FCP Procedure Encoding

$P(\dots) \text{ :- } g_1, \dots, g_n \mid \text{true.}$
 $P(\dots) \text{ :- } g_1, \dots, g_m \mid q_1(\dots).$
 $P(\dots) \text{ :- } g_1, \dots, g_k \mid q_1(\dots), q_2(\dots), \dots, q_n(\dots).$

Figure 4.10: FCP Clause Types

clause guard primitives, and let q_i denote user defined predicate relations. The first clause is called *halting clause*, the second *iterating clause* and the third is referred to as *spawning clause*.

All clause types are compiled in the following way. First, a set of unify and get instruction types denote clause-head unification. Following this, are the clause guard calls. If any of the above instructions fail, the clause at label FL is selected. After a successful clause-try, the commit instruction denotes the clause-commit phase of goal-reduction. This part of the clause level encoding with the abstract machine instructions is common to all three clause types.

The *halting* clause type denotes a clause with no body. In this case, the halt abstract instruction terminates the goal reduction. At this point, the goal structures of the terminated goal may be collected.

For the *iterating* clause type, the arguments of the new goal are created using a sequence of put and unify instructions. This is followed by an `iterate(Goal)` instruction. This instruction is actually an optimization of the `spawn` instruction used in the *spawning* clause type. That is, `iterate` assumes that the newly created arguments reuse the existing goal record and thus execution continues with the same goal record.

Spawning more than one goal in the the body of the committed clause is encoded by successive put and unify instructions followed by `spawn` instructions. The `spawn` instruction implements the scheduling of the newly created goal at the end of the active goal queue. After the arguments of the last goal are spawned the `iterate` instruction continues program execution with one of the newly spawned goals. In Figure 4.11 we show the encoding of the three FCP clause types.

4.2.4 Performance of Abstract Machine Implementation

A preliminary evaluation of the program execution time was performed for small FCP test applications executing on the VAX 11/750 host machine. For example, the abstract machine implementation executed the *naive reverse(100)* program at a rate of 2K LIPS, which is a six-fold improvement compared to the interpreter implementation on the same host machine. A peak performance of 10K was estimated for the trivial case of the $(p :- p.)$ iteration.

Even though a complete and thorough comparison of the two FCP sequential implementations was not performed, in general it is expected of the FCP abstract

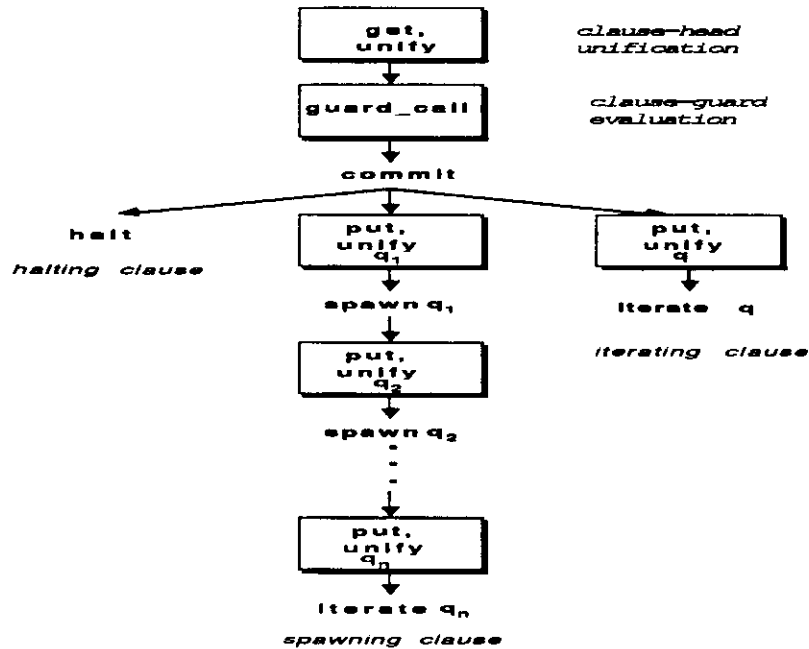


Figure 4.11: Compiling FCP Clause Types

machine implementation to perform better than the FCP interpreter (as the simple case of the naive reverse program may indicate). However, the real benefit of a compiler based abstract machine implementation is the potential for compile time analysis and optimizations that result in reduced program execution time. One important compilation technique called *Decision Tree Analysis*, is described in the following subsection.

4.2.5 Compiler Optimizations

A compiler for a subset of the FCP language is described in [Klig87]. The main concept introduced is the method of *Decision Tree Compilation*. Compared to the compilation approach used in the abstract machine implementation, the main difference is how the clause-tries in the procedures are compiled.

Let us consider the representation of a procedure with C clauses shown in Figure 4.12(a). The compiled program is stored as a sequence of clause-tries separated by the `try_me_else(L)` instruction (see Figure 4.9). Therefore, the clauses are tried sequentially. Within a clause, the arguments are tested for unification,

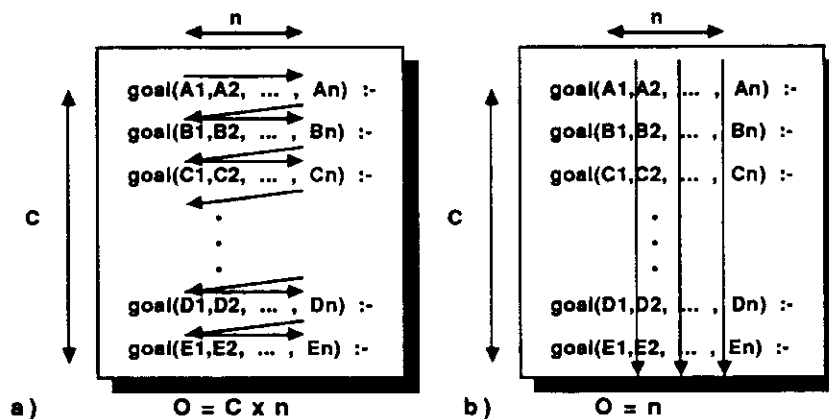


Figure 4.12: Decision Tree Compilation

again in a sequential manner, this time from left to right. If we consider that a procedure has C clauses and N arguments, then in the abstract machine, the upper bound of the number of arguments tested for unification is equal to $C \times N$.

As a result of the FCP programming style, one may note that, in a *typical* procedure, it is common for the same argument to be in several clauses. Using the abstract machine compilation approach, repetitious argument testing is performed. Moreover, if two clauses differ in only one argument, and the first one fails to match the calling goal, checking the same arguments of the next clause is redundant. It is quite clear that a compiler that performs inter clause-try optimizations is required.

In the *Decision Tree Compiler*, the complete procedure is compiled to a *decision tree* where the first step is to check whether the first argument matches any of the corresponding clauses. Since with the first test some of the clauses may be eliminated from further checking, the following argument checks may involve only a subset of the clauses. Finally, a committing clause is found within at most N argument checks. The main advantage is that all of the argument data types are known at compile time; therefore, the decision tree is formed at compile time and used to implement clause-head unification.

The *vertical* rather than *horizontal* clause selection is represented in Figure 4.12(b). The effect is that with this approach, all of the clauses in the procedure are *concurrently* performing a clause-try, rather than performing them sequentially. The decision tree takes the format of an interleaved argument test and

multiway branch. The multiway branch depends on the number of different data types defined.

One drawback of this compilation technique is that it may possibly generate large code size. This is due to the potential for an *explosion* of decision tree possibilities, that is, branches. However, in practice, preliminary measurements using simple test programs indicate a potential for performance improvements [Klig87].

4.3 A Distributed Implementation of FCP

In this section we describe two distributed implementations of FCP. The first is an extension of the FCP interpreter implementation described in Section 4.1 and the second is a compiler based distributed abstract machine implementation. In both implementations it is assumed that the execution environment consists of a set of parallel processors, each with only local memory and with direct communication links to a small number of neighboring processors. For example, a *hypercube* or a *torus ring* multiprocessor configuration would fit the above description. The main reason for considering a distributed system for the execution of FCP was the long term objective to implement FCP on a very large and scalable processing network. From this point of view, a shared-memory multiprocessor architecture was considered unsuitable. Before we consider the two distributed implementations of FCP, we first consider some common requirements.

Atomic Transactions

Since FCP is a committed-choice concurrent logic programming language with atomic unification, the distributed implementation must support the capability of atomic transactions across processor boundaries. In other words, since unification in FCP is performed as an atomic operation all the data structures used during a goal reduction must be exclusively accessed by the processor performing the goal reduction.

Mutually exclusive access requires a processor locking capability at the data structure level. Since the *logical variable* is the only data structure that can be assigned a value, that is modified, locking must be performed at the level of the logical variable. Together with the variable locking capability, a mechanism for preventing multiple writers from getting into a *dead-lock* situation should be

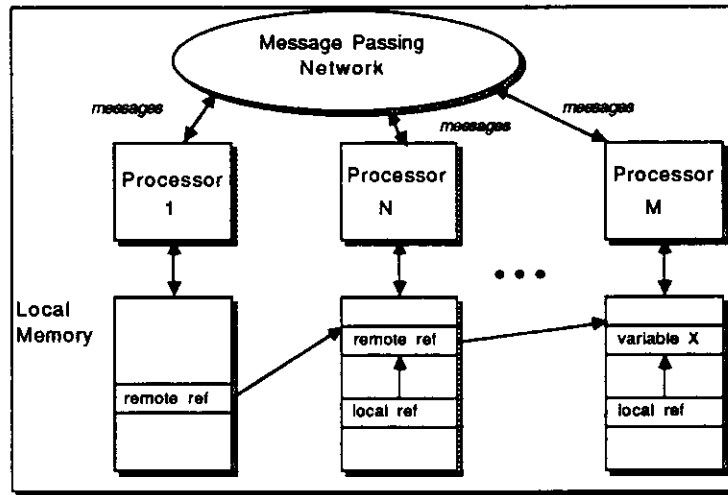


Figure 4.13: Remote References in a Distributed Execution Environment

ensured.

Shared Data Structures

A mechanism for accessing non-local shared data must be integrated into the FCP primitive operations. In both distributed implementations of FCP, a new data type called *remote reference*, allows program data structures to be shared by multiple processors, thus transcending the local memory boundaries and implementing a global memory address space. The remote reference contains the following information: remote reference tag RRef, remote processor identification number PID and a remote processor memory address A. In Figure 4.13 we show several remote references across multiple processors denoting the same logical variable. One should note that in the distributed implementation there is always only a single occurrence of a particular logical variable and all other occurrences are local or remote references to the variable.

Distributed Control Algorithm

The FCP distributed implementation consists of a distributed run-time environment and control mechanism that must ensure the same language semantics as the single-processor implementation. The distributed control algorithm de-

defines the sequence of control steps taken when a remote reference is encountered during execution. If a remote reference is not encountered, program execution proceeds as in a single-processor implementation. However, since there is no shared memory, when a remote reference is encountered the only way to access the remote structure is to send a message to the referenced remote processor. The processor receiving the message then responds accordingly thus maintaining a distributed access control of shared data structures. We now consider first the FCP distributed interpreter followed by the FCP distributed compiler based implementation.

4.3.1 FCP Distributed Interpreter

A distributed FCP interpreter written in Occam is described [Bar-86]. Occam was chosen with the intention of implementing FCP on a network of InMos Transputer processors. The main control algorithm and the run-time environment of the interpreter are defined as an extension of the FCP interpreter implementation described in Section 4.1. The main extension to the sequential run-time environment are two message queues QIn and QOut used to buffer incoming and outgoing messages. Since the processing of incoming messages may alter the program data structures, this is performed between successive goal reductions so as to ensure atomicity.

The FCP distributed unification control mechanism used in the interpreter implementation represents the first version of the algorithm subsequently improved. We therefore discuss some of the preliminary observations and performance results of the interpreter implementation and describe the improved control algorithm in Section 4.3.2.

4.3.1.1 Reported Preliminary Performance Results

Preliminary performance measurements of the FCP distributed interpreter implementation, using small test applications, indicate small speed-up values. For a 2x2 grid architecture a speed-up between 1 and 2 is reported, and for the 4x4 processor grid architecture a speed-up of 1 to 4.5 was obtained. The following drawbacks of the distributed FCP interpreter implementation were cited as the reasons for the poor performance:

- *Idle Time.* In most test applications the processor *idle time* was up to 30% of the program execution time. The idle time is due to the communication protocol.
- *Communication Overhead.* In all applications, the inter-processor communication overhead was unacceptably high. In some cases the ratio of reduction time versus communication time was equal to one, meaning that 50% of the execution time was spent performing operations related to inter-processor communication.

Some of the reported performance results may be explained by the inefficient implementation of FCP in Occam. That is, Occam was the language used to implement the FCP interpreter, with the intention of executing on a network of Transputer machines. Since the actual Transputer machines were not available, Occam was implemented on the Vax 11/750 machine. The sequential implementation of the FCP interpreter written in Occam executed 3 times slower than the same interpreter written in Pascal. However, the distributed implementation timing results were obtained by simulating execution time on the 2x2 and 4x4 grid processor architectures.

4.3.2 FCP Distributed Abstract Machine

A compiler based distributed abstract machine implementation of FCP, as well as a prototype implementation on the ipSC Hypercube architecture, is described in [Tayl89]. We now briefly describe only the main features of the distributed unification algorithm.

4.3.2.1 Distributed Unification Algorithm

If all data structures are local during goal reduction, the distributed unification algorithm implements the single-processor abstract machine control mechanism. Inter-processor messages are sent as a result of remote references to shared data structures. Distributed unification may require either read or write access to the remote data structure. For read access, no locking is necessary. However, a write access request must lock the variable prior to the assignment. Moreover, exclusive lock requests must be obtained for all data structures that are modified during a goal reduction. The locks are released upon goal commit.

The FCP unification algorithm for the implementation of atomic transactions in a distributed environment, is defined using Read, Value, Lock and Grant messages. We describe how the messages are used in the following two scenarios:

- Read access to a remote reference
- Write access to a remote reference

Read-Remote Reference

The Read message is sent to the remote processor when an attempt is made to read the remote reference, that is, when goal reduction requires the value of a data structure located at a remote processor. The response of the processor that receives the Read message depends on the data type of the referenced structure. If it is itself a remote reference, the message is forwarded to the new destination. If the address refers to a ground term, the data structure is packed and sent to the processor that originated the message. This is performed using the Value message. However, if the referenced data is a variable, then a mechanism is implemented to notify the requesting processor whenever the variable receives a value. This is performed in the same way as a goal suspension. That is, a *broadcast note* is enqueued onto the variable suspension queue. When some goal reduction commits by assigning a value to the variable with the broadcast note, a Value message is sent with the newly arrived data.

Write Remote Reference

A Lock message is sent when an attempt is made to write on a remote reference. This message requests exclusive access to the remote variable by attempting to lock it. In order to prevent dead-lock, processors are discriminated using priority numbers. A Lock request from a high-priority processor is considered a *high-priority lock*. In a way similar to the above *read remote reference* case, the Lock message is forwarded if it refers to another remote reference, and a Value message is returned if the referred term is ground.

The lock on a variable is immediately granted to a high-priority lock message using the Grant message. If there were any goals suspended on the variable, these goals are activated. The local variable is then converted to a remote reference, and the processor that originated the Lock message converts the remote reference

to a variable. In this way, variable migration is implemented in order to bring variables closer to processors that modify them.

In case a Lock message comes from a low-priority processor, the lock is granted if the variable is unlocked. Otherwise, the lock is deferred. The deferred lock request is granted only when the higher-priority processor relinquishes its lock on the shared variable.

4.3.2.2 Preliminary Performance Results

The performance study of the distributed FCP implementation was performed by executing five FCP benchmark programs and five selected program stereotypes on a four dimensional (d-4) Hypercube multiprocessor. In summary, three of the applications exhibited a speed-up between 9.4 and 12. The poor speedup of the other two applications (less than 3.5) is attributed to “low granularity and poor load balancing.”

Of the five selected stereotype program techniques, poor speed-up results are also reported. Two of the programs showed performance degradation when executed on two processors relative to the single processor implementation. Another showed a performance degradation of 3.8 on the d-4 Hypercube architecture relative to a single-processor implementation. One of the reasons for the poor performance is the overhead of locking shared data structures.

4.4 Summary

In this section we described four different implementations of FCP on general-purpose machines. The FCP interpreter represents the first prototype implementation that evolved into the subsequent implementations. The sequential abstract machine based on the abstract machine for Prolog opened the door to compiler optimizations. For example, decision tree compilation may reduce the overhead of unsuccessful clause-tries. Whereas the performance of the FCP interpreter was several hundred LIPS for a *naive* program, the same program showed a six-fold improvement in the abstract machine implementation. Even though a performance analysis was not made, it is reasonable to assume that the abstract machine implementation will outperform the interpreter by an order of magnitude. Further improvements are expected with the new compilation technique.

Two distributed implementations of FCP were described. The first is based on the FCP interpreter and the second is based on the FCP abstract machine. In both cases the implementors of the language, after performing preliminary measurements, quote the overhead of communication as the main reason for the poor performance of the distributed implementation. Even though the unification algorithm described in [Tayl89] is claimed to be *simple*, the overhead of locking required to maintain atomic transactions in a distributed system affects program execution time significantly.

Therefore, the degree of sharing amongst concurrent goals in an FCP program is perhaps more suitable for a tightly coupled rather than loosely coupled multiprocessor system, at least at the size of the multiprocessor systems considered, (note, large processing networks and large applications were never evaluated). In this thesis we will consider an execution model and special-purpose architectural support and evaluate the suitability of larger applications to make use of parallelism in a shared memory environment.

CHAPTER 5

Special-Purpose Architectural Support: Design Approach and Results of Analysis

This chapter is divided into six sections. In the first section, we define what is meant by *architectural support* for FCP and we discuss the main motivation as well as the scope of the support. In the second section we propose an approach for the design of a special-purpose processor architecture based on previously reported bottlenecks observed in existing implementations of FCP. The tradeoffs involved in the design and analysis are pointed out. The analysis approach, described in the third section, is based on empirical evaluation methods. For this purpose we select benchmark programs that represent a specific system workload. The workload is described in the fourth section. The main emphasis of this chapter is the performance modeling and analysis of potential implementation bottlenecks, discussed in the fifth section. Finally, a general goal reduction model is described in the sixth section. A multi-functional unit processor organization is discussed as a way of providing special-purpose support for the implementation of FCP. The specific processor architecture organization and execution model is described in Chapter 6.

5.1 Architectural Support

As it was described in Chapter 4, current implementations of FCP execute on general-purpose processors, either sequentially or distributed in a medium-scale multiprocessor environment. The performance of these implementations does not adequately meet the users' needs. This may be accounted for in the following:

- The field of concurrent logic programming is a recent one, so that many issues remain unsolved and unoptimized. For example, new compilation techniques are currently being researched, promising significant performance improvements. Improved algorithms for a distributed implementation is another research topic under investigation.

- The execution model of concurrent logic programming languages is particularly unsuitable for emulation and execution on existing general-purpose machines.

We are interested in analyzing ways of improving the performance of FCP implementations in addition to the advances in compilation techniques and other system software issues. We thus consider *architectural support* for FCP implementations. Architectural support for the execution of a high-level language implementation like FCP, can be considered in either of the following two forms:

- *General-purpose architectural support* is an effective way of improving performance using existing general-purpose components. For example, increasing main memory size, adding a general-purpose memory cache, a functional co-processor or upgrading the existing machine with a faster general-purpose processor is considered general-purpose architectural support.
- *Special-purpose architectural support* involves the addition of components that are specifically *designed* for the performance improvement of the target language implementation. For example, a specialized cache with a cache-policy or organization designed to match the specific characteristics of the language is considered special-purpose support. In other words, the design of special-purpose architectural support requires the characterization of system behavior which is then used in the process of design.

Motivation

The main motivation to consider special-purpose architectural support for FCP is the potential for improvements in system performance beyond what is possible using general-purpose architectural support. The design of special-purpose support is more expensive than using existing general-purpose support. Therefore, a cost-effective improvement of system performance may involve a combination of architectural supports according to a specific cost function. We now discuss the design approach used to propose a special-purpose processor.

5.2 FCP Processor Design Approach

In this section we discuss some of the tradeoffs involved in designing a special-purpose processor architecture for FCP. Instrumental in the design approach is the process of system analysis. We describe our approach and compare it to previously reported approaches.

5.2.1 From Operational Semantics to Machine

It seems only reasonable that the process of designing a special-purpose processor for FCP should start from the language itself, that is, its semantics. In particular, one could use the operational semantics of the language, defined as an algorithm that describes the machine independent execution of programs in that language. For the programming language FCP, the operational semantics is fully described in [Mier85].

Hypothesis Driven Design Approach

Ideally, starting from the operational semantics of a high-level language, one method for the design of a processor architecture may be described as a search in the space of architectural alternatives. At each step of the search, the change in system cost and performance is evaluated according to a set of cost and performance criteria. A cost-effective design is then represented as a region that satisfies the set of performance and cost constraints originally defined as the design goal.

In practice, however, the number of architectural alternatives even for a simple design is so large that any meaningful systematic approach is out of the question. By limiting the number of alternatives to a small subset, the prospects of a systematic design become more reasonable, but at the expense of finding a cost-effective solution. We now discuss a less systematic but practical design approach.

Using an existing implementation of FCP, our approach for the design of special-purpose architectural support consists of first obtaining high-level program characteristics of a specific system workload. Based on these results, a *hypothesis* is made regarding the potential system bottleneck. A performance model for the selected phenomenon is defined, evaluated and the hypothesis is

thus *verified*. This process is repeated for other suspected bottlenecks.

It is very important in this design approach that the dependency of the obtained system parameters on the language implementation be well understood, and where possible abstracted. The tradeoffs involved in obtaining the program parameters are discussed in the following section.

5.3 FCP Implementation Analysis Tradeoffs

In Chapter 4 we reviewed the compiler-oriented FCP implementation based on the emulation of a high-level abstract machine. However, the sequential abstract machine represents only one implementation of the FCP operational semantics. Alternative abstract machines with different data structures and instruction sets could be proposed. In Figure 5.1 we symbolically represent the implementation of the FCP operational semantics using the sequential abstract machine, whereas possible alternative abstract machines are depicted as shaded boxes.

One way of executing FCP programs on existing *real* machines is to emulate the abstract machine using an emulation language that executes on the host machine. The choice of the emulation language is usually made based on issues such as portability, programmability and implementation efficiency. Which emulation language is more suitable for emulating the high-level language abstract machine is an important question, but is beyond the scope of this report. Also shown in Figure 5.1 are some of the emulation language and physical machine choices made in environments where the FCP language is used. Whereas in all cases the emulation language is the same (C programming language), the host machines vary (VAX, SUN, CCI). The use of different emulation languages is also represented using *shaded boxes*. Also shown in Figure 5.1 is the special-purpose processor architecture as well as the execution model.

For a given system workload, one may define specific measurement tools to obtain a set of parameters that will characterize the language execution at various implementation levels [Ferr89]. For example, the FCP interpreter could be enhanced to capture the algorithmic behavior of the semantics. On the other hand, hardware tools could be used to describe the memory reference behavior or the processor register usage of the particular physical machine.

The choice at which level to characterize the implementation is a very impor-

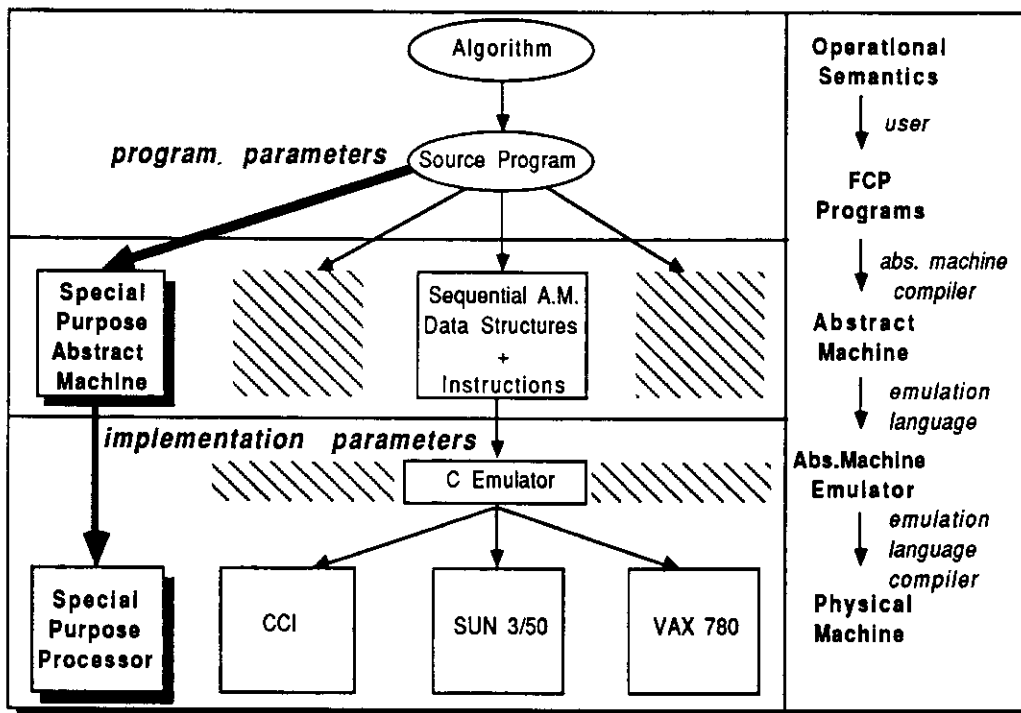


Figure 5.1: FCP Implementation via Abstract Machine Emulation

tant one and depends on the design objective. A characterization of a language implementation that does not take into account the purpose of the measurements may result in erroneous design choices. We recognize the following two objectives:

- Improvements to the specific implementation
- The design of a special-purpose implementation

To improve the system performance of a particular implementation using either *general-purpose* or *special-purpose* architectural support, analysis is appropriate at all levels of the implementation, since they are characteristic of the specific execution environment. Other issues such as the availability or cost of the measurement tools may be the deciding factor.

However, if the objective is to obtain implementation characteristics for the purpose of designing a special-purpose processor, it is essential that they be either abstracted from the current implementation or shown to be true in other implementations. In the case of the compiler based implementation of FCP, one would want to consider characteristics that are invariant to the selection of the abstract machine implementation, emulation language and host machine.

For example, high-level program characteristics such as complexity, termination detection, dead-lock detection and prevention, non-determinism etc., could be analyzed by symbolic interpretation using another high-level language implementation. To interpret and analyze logic programs, Prolog is often used.

Unfortunately, the number of features that can be captured at the symbolic interpretation level is usually smaller than at a lower level. On the other hand, as measurements are obtained at the lower level so they become more dependent on the actual implementation. Therefore, the essence of the described tradeoff is between *the level of detail versus the generality of the results*.

5.3.1 Proposed Implementation Level Analysis

The main motivation for the considered implementation analysis is the design of a special-purpose processor for FCP. We propose an implementation analysis and design approach based on the following two features:

- Hypothesis driven empirical characterization of program execution at the abstract machine level.
- Low-level implementation characterization of the specific special-purpose processor architecture for FCP.

The abstract machine characterization is independent of the selected emulation language and the underlying host machine. However, it does depend on the selected abstract machine data structures and defined abstract machine instructions. These results describe the high-level program behavior and are used to suggest the top-level processor organization. Low-level implementation parameters such as timing measurements or memory reference behavior are captured only after the complete specification of the processor, which includes the processor instruction set.

Analyzing program execution at the sequential abstract machine level also has the following advantages:

- All implementations of the FCP programming language currently in use are based on the emulation of the sequential abstract machine.
- A full compiler to the abstract machine instructions is available.
- A significant number of large FCP program applications is available in the existing abstract machine environment.
- A complete development environment, the Logix operating system, has been developed using the proposed abstract machine organization.
- As of yet, no other sequential abstract machine for FCP has been proposed.

To obtain FCP implementation characteristics at the abstract machine level, a new instrumented version of the existing abstract machine emulator has been created. This new emulator, called *slogix* (statistics logix), contains extra data structures such as tables and counters for the purpose of collecting statistics. Logically, FCP programs execute as if executing in the original emulation environment, only 3 to 5 times slower. All of the results presented in this thesis are derived using *slogix*.

In Figure 5.2 we show the approach used for program analysis at the abstract machine level. Existing, FCP applications are used as benchmark programs.

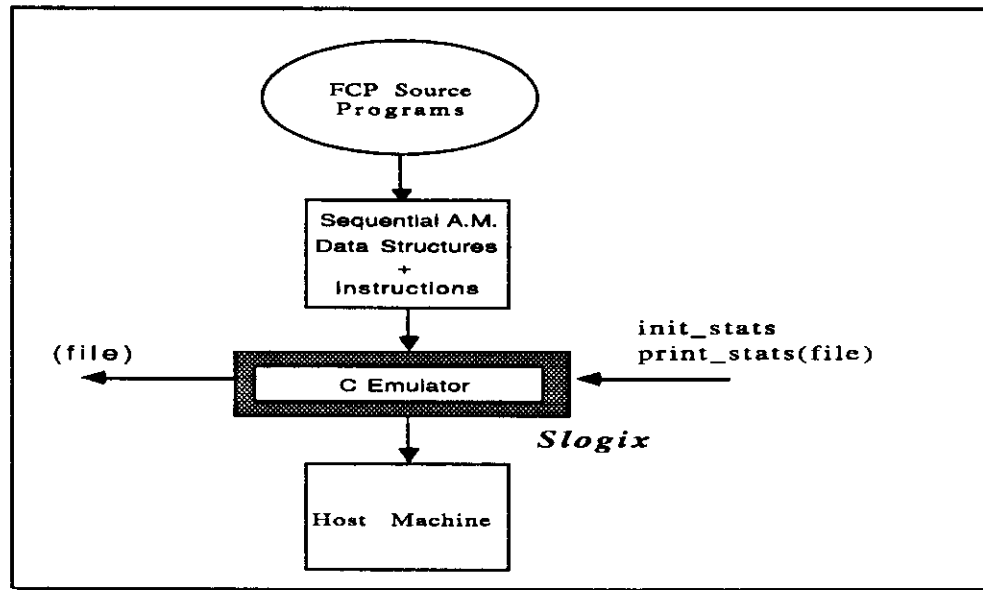


Figure 5.2: FCP Program Analysis Approach

They are compiled, using the FCP abstract machine compiler, to abstract machine instructions. The compiled programs are then executed by the *slogix* emulator. Additional features are also introduced to aide program analysis. These are the *machine#request(init_stats)* and the *machine#request(print_stats,file)* logix commands. Before executing a program the *machine#request(init_stats)* command initializes all the data structures used for statistics gathering. This is necessary since Logix is also written as an FCP application and the effect of its execution on statistics is also recorded unless initialization is performed prior to a specific benchmark analysis. The *machine#request(print_stats,file)* is used to store the results in *file*.

5.3.1.1 Previous Approaches

Profiling Logix with *gprof*

An attempt to characterize FCP program execution has been proposed in [Gino87]. The motivation for the analysis was similar to ours, that is, the design of a special-purpose processor architecture. However, in this approach, program execution is analyzed at the host machine level using existing profiling tools of

the emulation language and the operating system. Specifically, the Unix *gprof* is used to determine time consuming events in the language emulator. This approach suffers from the following drawbacks:

- The program execution analysis is dependent on the emulation language. For example the results may differ depending on whether C or Pascal is used.
- The program execution analysis is dependent on the host machine. Versions of the emulator exist for the Vax, CCI, SUN ...
- The program execution analysis is dependent on the implementation of the emulation language on the host machine. For example, the results may differ depending on the emulation language compiler used.
- The program execution analysis results depend on the use of the emulation language. Whether a *macro* rather than a *function call* is used may alter program characteristics.
- Using timing facilities provided by an operating system commonly incur a margin of error. If the profiling analysis is used to repeatedly evaluate the duration of an event that is of the same order of magnitude as the incurred error, the accumulated error may be too high.

In other words, this analysis approach may be suitable to capture the characteristics of the specific implementation (for example the emulation language C was used and the host machine was the VAX) but not for the design of a special-purpose processor. Moreover, the measuring tools used in this approach do not allow for an accurate characterization of a specific implementation.

Applying the profiling technique on the Logix operating system using the above described approach, it is claimed that 22% of the execution time is spent dereferencing, 17% copying, and 15% of the execution time performing type identification. These results are then used as a basis for the design of specialized hardware support for FCP, described in [Hars88].

Performance Analysis of FGHC on the Sequent

In [Tick88], a detailed analysis of FGHC execution in a parallel programming environment, namely the Sequent multiprocessor architecture is described. The

main motivation for the performance analysis is to compare the execution of an AND-parallel, committed-choice logic programming language like FGHC with the execution of an uncommitted-choice OR-parallel logic programming language Aurora, running on the same multiprocessor machine. The program analysis is performed at both the abstract machine level, called *high-level* analysis, and the physical machine level referred to as the *low-level* analysis. In addition, the raw timings of the two parallel systems are compared using selected benchmark programs.

Our work differs from that presented by Tick in the following way. Since the concern in [Tick88] is to characterize and optimize execution of a specific execution model and compare it to another model, most of the analysis is appropriately performed at the physical machine level. According to the classification earlier described, this type of analysis is concerned with *special-purpose* support for an implementation on a general-purpose machine. Since we are concerned with characterizing FCP program execution for the sake of proposing a *special-purpose* processor architecture, it is important that the analysis be machine independent. Therefore, all of our analysis is at the abstract machine level and not the physical machine level. To extrapolate low-level machine program execution analysis from one machine to another requires justification. This is particularly true if a new processor architecture is under consideration.

5.4 Empirical Analysis of a Specific System Workload

The FCP implementation characteristics used in this thesis are derived empirically by executing FCP benchmark programs. The selection of sample programs for the purpose of performance analysis is a critical step. We now characterize the *system workload* used for empirical performance analysis.

System's Development Workload

For the FCP program analysis we have selected 7 large FCP programming applications. For each application, the following holds:

- All of the programs are authentic, *unaltered*, FCP programs that were not optimized for the purpose of benchmarking. Also, the programs were written by various programmers, thus exhibiting a variety of programming

styles.

- Since program analysis was the main motive for benchmarking, the programs were not selected for their performance, but rather for their characteristic behavior.
- All FCP programs are real applications used repeatedly within the Logix development environment.
- All of the selected FCP programs are large in terms of the number of high-level source code lines. Also, they all run for several hours. This is important in order to get a reasonable average of program behavior.
- The selected programs exhibit considerable program complexity and make use of programming techniques representative of FCP.

Therefore, the selected programs are representative of the FCP programming environment currently under development. If there is one drawback of the selected programs, it is that they are specific of a single development environment. That is, due to the lack of a large spectrum of user applications, the selected programs are limited in their scope of application to system applications for program development. We thus refer to the selected FCP programs as characteristic of a *System's Development Workload*. Other workloads specific of alternative application areas may be defined and may thus exhibit different behavior.

5.4.1 Selected FCP Benchmarks

In Table 5.1 we show some of the features of the selected FCP benchmark programs. Some applications were written in FCP using user annotations of variables in order to model inter-goal communication and synchronization; we denote these as written in *fcp(?)*. More recently, applications have been written using input unification in the guard to model goal suspension; we denote these applications as *fcp(:)*.

Also shown in Table 5.1 are the modes of execution of the benchmark programs under the Logix system. That is, a program can execute in *trust* or *interpret* mode. In trust mode, program execution proceeds as if each goal reduction may either succeed or suspend. A failure in this mode leads to program failure. In order to analyze program execution and detect failure, the program must be compiled using the interpret mode declaration. For comparison, one of

Info	FCP Benchmark Programs						
	Comp.	Sim1	Sim2	Debug	Solver	Distr.	Logix
Lang.	fcp(?)	fcp(?)	fcp(?)	fcp(:)	fcp(?)	fcp(:)	fcp(?)
Mode	trust	trust	interp	trust	interp	interp	trust
Lines	3885	2066	2066	2000	676	2097	10000
Red.	7075380	3009280	7722376	1593286	409075	259283	1481900

Table 5.1: FCP Benchmark Information

the benchmark programs (the FCP Processor Simulator) is benchmarked in both trust and interpret mode. A program running under interpret mode generally runs 3 to 5 times slower than in trust mode.

One should note that the selected FCP programs are all large. This is quite important, and is one of the main drawbacks of previously used benchmarks, as we will discuss shortly. The smallest application is the program analysis called Solver, which is 676 lines of FCP code. As far as the number of goal reductions performed, we see that the average number is of the order of several million.

We now briefly describe each of the used FCP benchmark programs. The program source is not listed here due to their large size, but can be requested from the corresponding authors, as shown in Table 5.1.

FCP Compiler

The FCP Compiler consists of 3885 lines of FCP code [Hour86]. It translates FCP source code to the sequential abstract machine instructions. The benchmark program consists of compiling the complete Logix Operating system which consists of 10000 lines of FCP code divided into 12 different modules.

FCP Processor Simulator

The FCP Processor simulator consists of 2066 lines of source code. The program specifies the architecture of the special-purpose processor for the execution of FCP, described in this thesis. The processor consists of concurrent functional units that cooperatively perform goal reduction. A full description of the proposed special-purpose FCP processor is described in Chapters 6 and 7. The actual benchmark consists of running the processor simulator as it interprets a

hand-compiled version of small application programs.

FCP Debugger

The FCP Debugger is a program application used for debugging FCP programs. It is written as an FCP meta-interpreter, and consists of over 2000 lines of FCP code.

OR-Parallel Prolog Solver

The OR-parallel Prolog solver performs an abstract interpretation of a program with a goal. The algorithm itself is not practical for application to larger programs but is the basis for flow analysis, mode analysis and type inference. The algorithm computes the least fixed point of a function, based on the OLD T proof strategy of Tamaki and Sato.

FCP Distributed FCP Implementation Simulator

The FCP Distributed Implementation Simulator simulates the execution of FCP over a system of distributed processing nodes. FCP programs are annotated with inter-processor communication pragmas. The actual benchmark program executes the *reverse* of a list on a ring network of 20 processors.

Logix Session

One of the most commonly used applications of FCP is the Logix operating system and development environment. It consists of over 10000 lines of FCP code written by various programmers. To benchmark the use of the Logix operating system, a script was kept of a “typical” system workload. This consisted of a sequence of compilations, program runs with errors, debugging sessions, type checking and so on.

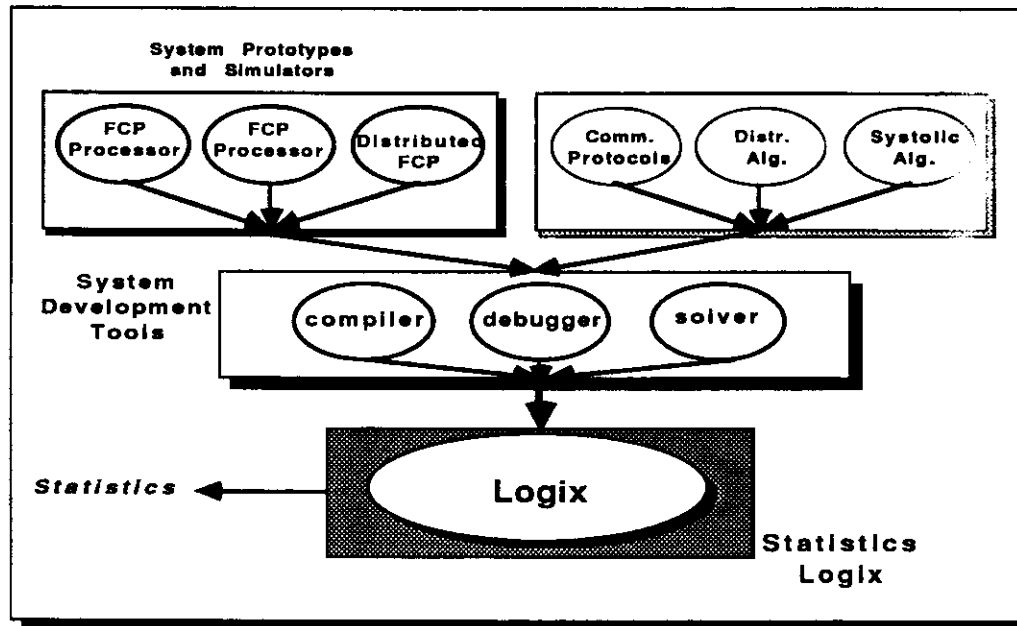


Figure 5.3: System's Development Workload Session

5.4.2 A Workload Session

A typical System's Development Workload session is described as follows. FCP programs for system prototyping and simulation are developed under the *Logix* development system. Two different simulation environments are being modeled. One is a concurrent, multi-functional unit environment for the execution of FCP called *Simulator1*, and the second is a simulator of a distributed FCP implementation executing on a 20-node ring architecture, called *Distribute*. A second version of the *Simulator1* program called *Simulator2* (compiled in interpret mode) is debugged using the *Debugger* program. All programs are compiled using the *FCP Compiler*. Program flow analysis using abstract interpretation is performed using the *Solver* program.

The 7 selected FCP programs and their use in the System's Development Workload session is shown in Figure 5.3. The characteristics of the programs are equally weighed even though they vary in size and execution time. This is because we view them as representing different types of workloads, and we are interested in their average.

Also shown in Figure 5.3 is the use of other possible workloads besides the System's Development Workload. For example, the use of FCP in modeling

communication protocols and distributed algorithms has been suggested [Shaf84] [Hell83].

5.4.3 Previously Used Benchmarks

The benchmark programs used in this thesis differ significantly from the previously used benchmarks. In [Gino87], small applications such as *string append*, *string reverse* and *quicksort* are used as well as the larger *parse* program. The largest program size was 260 lines of FCP code. These programs are not representative of a specific FCP application domain.

In [Tick88], larger program applications are used. However, the author correctly acknowledges that the main drawback of the analysis is the use of "small symbolic manipulation problems". On the other hand, the benchmarks were used as a means of comparing two different language architectures, and not so much to characterize a single language implementation. Therefore, the relative comparison makes the analysis more meaningful and useful.

Another set of benchmarks was used in [Tayl89] to evaluate the distributed unification algorithm described in Chapter 4. Five program stereotypes were used as characteristic of a particular program behavior. These programs are small and are used to model typical inter-process communication patterns and not overall system performance.

5.5 Analytic Performance Evaluation of Hypothesized Bottlenecks

We now consider the following potential implementation bottlenecks, as reported by various researchers:

- Overhead of Clause Selection
- Frequent Goal Suspension and Activation
- Pointer Dereferencing
- Clause Trailing

For each case, we define and analyze a performance model for a specific range of system parameter values. The model assumes a sequential execution environment for FCP based on the abstract machine described in Chapter 4. Before we discuss each of the reported bottlenecks, we first describe the general approach used in the process of analytic modeling and performance analysis.

5.5.1 Analytic Performance Models

One of the objectives and benefits of using analytic models for performance evaluation is the ability to inspect a wide space of performance characteristics using analytic means. To do so empirically requires the execution of numerous and lengthy simulation sessions, thus making this approach impractical. For the analytic model to be useful, one must be able to vary the system parameter values and evaluate their effect on system performance in a computationally tractable way.

In Figure 5.4 we symbolically denote the performance model as a set of functions or expressions defined over the domain of system parameters. We also show that the system parameters belong to two distinct groups: one group characterizes *implementation-dependent* parameters that depend on system organization or architecture and the other group characterizes *implementation-independent* program parameters. By separating the two sets of parameters one is free to independently abstract program behavior in terms of the proposed parameters. In general, characterizing program behavior is very difficult. However, we are concerned with modeling only specific aspects of program execution, and therefore

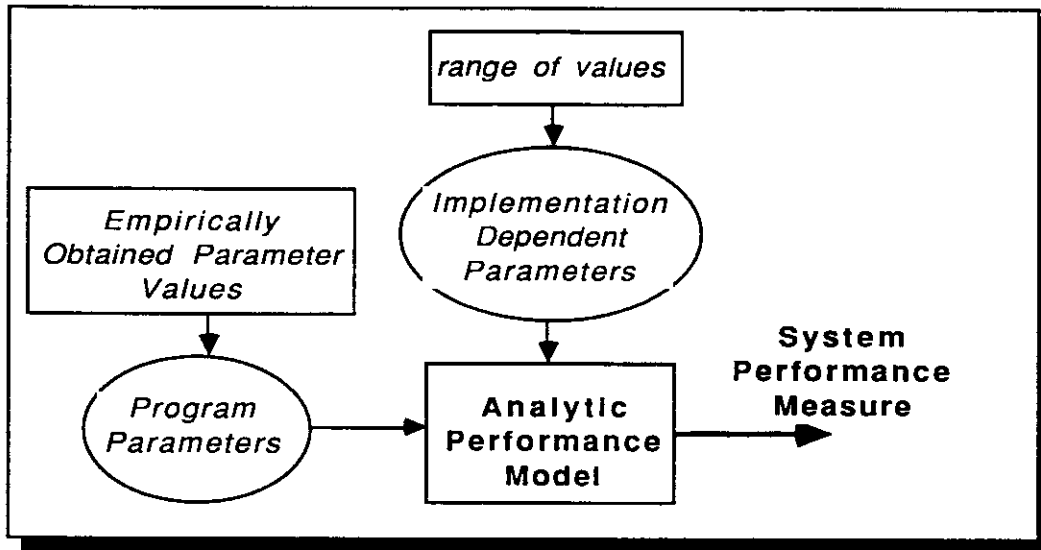


Figure 5.4: Analytic Performance Models

are able to characterize program behavior in terms of a few well defined program parameters.

We present the performance analysis in the following three steps.

1. **Performance Measure:** We define a common system performance measure for all the suspected implementation bottlenecks. It is important to precisely identify and define the performance measure since it determines what to model and evaluate during the performance analysis.
2. **Performance Model:** A performance model defines a relation between the performance measure and other parameters in the model. We group the parameters into *program* dependent and *implementation* dependent parameters. Since we are interested in a machine independent performance analysis, we consider the machine dependent parameters as variables in the model.
3. **Performance Model Analysis:** The analysis of the performance model consists of inspecting the range of performance measure values for a specific range of program parameter values and machine dependent parameter values. In our analysis, the program parameters are obtained using the *slogix* system executing the previously defined benchmark programs.

General Performance Measure

Let \bar{t}_f represent the average execution time of function f during goal reduction in a specific sequential implementation of FCP. Let \bar{t}_r denote the average execution time of the *remaining* part of goal reduction in the same implementation environment. Therefore, the average execution time of a goal reduction \bar{t}_{red} is represented as:

$$\bar{t}_{red} = \bar{t}_f + \bar{t}_r \quad (5.1)$$

We now define the *relative execution time* of function f as:

$$O_f = \frac{\bar{t}_f}{\bar{t}_{red}} \quad (5.2)$$

that is,

$$O_f = \frac{1}{1 + \frac{\bar{t}_r}{\bar{t}_f}} \quad (5.3)$$

where $\bar{t}_f \neq 0$.

We specifically do not refer to the *relative execution time* of function f as the overhead associated with the function execution since it is not clear at this stage what are the practical limitations in the function implementation, and what is truly an overhead. What we can say, is that the relative execution time corresponds to the overhead of function execution with respect to an idealized system in which the average function execution time is $\bar{t}_f = \varepsilon$, where $\varepsilon \rightarrow 0$. We now consider each of the previously hypothesized implementation bottlenecks.

5.5.2 Redundant Clause Selection

In the sequential abstract machine, all clause-tries are performed sequentially, in textual order. We refer to all clause-tries that fail and all clause-tries that suspend but do not result in goal suspension as *redundant clause-tries*. In other words, the number of redundant clause-tries reflects the difference between the clause-selection strategy used in the program and a *perfect* clause-selection algorithm. We now define the redundant clause-try performance model and parameters, followed by the results of analysis.

5.5.2.1 Performance Model

Let us represent the average execution time of a goal reduction \bar{t}_{red} as follows:

$$\bar{t}_{red} = F_{ct}^r N_{ct}^r \bar{t}_{ct} + (1 - F_{ct}^r) N_{ct}^r \bar{t}_{ct} + \bar{t}_r \quad (5.4)$$

where N_{ct}^r represents the average number of clause-tries per goal reduction, F_{ct}^r represents the fraction of *redundant* clause-tries, \bar{t}_{ct} the average clause-try execution time per clause-try, and \bar{t}_r the average *remaining* goal reduction execution time which includes goal suspension, activation, spawning and termination. Note that the term $N_{ct}^r \bar{t}_{ct}$ represents the average clause-try execution time per goal reduction. The relative execution time of redundant clause-tries is then represented as:

$$O_{ct} = \frac{1}{1 + \frac{(1 - F_{ct}^r) + \frac{\bar{t}_r}{N_{ct}^r \bar{t}_{ct}}}{F_{ct}^r}} \quad (5.5)$$

Let κ denote the ratio:

$$\kappa = \frac{\bar{t}_r}{N_{ct}^r \bar{t}_{ct}} \quad (5.6)$$

The performance parameter O_{ct} is now simplified to the following expression:

$$O_{ct} = \frac{1}{1 + \frac{(1 - F_{ct}^r) + \kappa}{F_{ct}^r}} \quad (5.7)$$

In this performance model, we consider the fraction of redundant clause-tries F_{ct}^r a program parameter and κ an implementation dependent variable. Note that κ represents the ratio of the total execution time of operations that are not part of a clause-try, relative to the total clause-try execution time.

To determine the fraction of redundant clause-tries, let CT denote the total number of clause-tries, CS clause-suspensions, CF clause-failures and CR successful clause-tries or reductions. The total number of clause tries CT is equal to the sum of the clause failures CF, clause suspensions CS and reductions CR. Since a clause suspension need not result in the suspension of the goal, we also consider the total number of goal suspensions GS.

Let us now consider all clause-tries that lead to (goal) reduction and all clause-suspensions that result in goal-suspension as *useful* clause-tries, and consider all clause-tries that fail and all clause-suspensions that succeed, as *redundant* clause-tries. We can now denote the total fraction of redundant clause-tries as:

$$F_{ct}^r = 1 - \left(\frac{CR}{CT} + \frac{GS}{CT} \right) \quad (5.8)$$

FCP Benchmark Programs							
	Compiler	Sim.1	Sim.2	Debug	Solver	Distr.	Logix
CT (10^6)	20.14	18.94	48.99	7.46	4.52	2.59	5.85
CR (10^6)	7.07	3.00	7.72	1.59	0.40	0.25	1.48
CF (10^6)	10.65	3.58	24.93	2.94	3.79	1.79	2.84
CS (10^6)	2.41	12.35	16.33	2.92	0.31	0.54	1.52
GS (10^6)	0.88	1.89	2.51	0.50	0.05	0.08	0.30
$\frac{CR}{CT}$	0.35	0.16	0.16	0.21	0.09	0.1	0.25
$\frac{CF}{CT}$	0.53	0.19	0.51	0.39	0.84	0.69	0.49
$\frac{CS}{CT}$	0.12	0.65	0.33	0.39	0.07	0.21	0.26
$\frac{GS}{CS}$	0.37	0.15	0.15	0.17	0.16	0.15	0.20
$\frac{GS}{CT}$	0.04	0.10	0.05	0.07	0.01	0.03	0.05
F_{ct}^r	0.60	0.74	0.79	0.82	0.9	0.87	0.69

Table 5.2: Clause-Try Statistics

where,

$$CT = CR + CS + CF \quad (5.9)$$

We now discuss each of the above performance model parameters in turn.

5.5.2.2 Performance Model Parameters

Redundant Clause-Try Frequency, F_{ct}^r

In Table 5.2 we show the total number of clause-tries (CT), clause-suspensions (CS), clause-failures (CF) and successful clause-tries or reductions (CR) found in the selected FCP programs. We also show the ratio of the successful clause-reductions to the total number of clause tries (CR/CT), the rate of clause-try failures and suspensions per clause-try (CF/CT) and (CS/CT) respectively. Since not all clause suspensions result in goal-suspension, (an alternative clause may succeed), we show the ratio of real goal suspensions versus total clause suspension, (GS/CS).

The total number of redundant clause-tries shown in Table 5.2 ranges from 60% in the case of the *Compiler* to 90% in the *Dist* program. The average value of redundant clause-tries is 77%. It is precisely this issue of the large number of redundant clause-tries that is addressed in the work in [Klig88a] using compilation techniques.

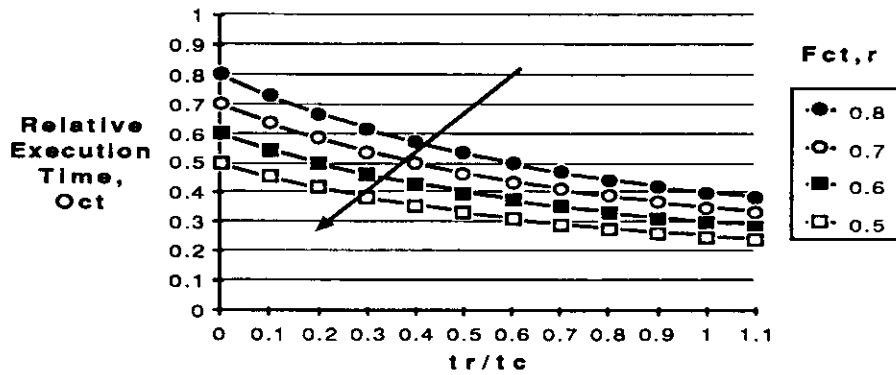


Figure 5.5: Redundant Clause-Try Relative Execution Time

Performance Model Variables

A single performance model variable, κ , is defined that represents the ratio between remaining goal reduction time and the clause-try execution time. The average clause-try execution time per goal reduction, denoted as $N_{ct}^r \bar{t}_{ct}$ includes all redundant clause-tries as well as clause-tries that lead to goal-commit or goal-suspend.

5.5.2.3 Performance Model Analysis

The average fraction of redundant clause-tries obtained from Table 5.2 is $F_{ct}^r = 0.77$. Since there are so many redundant clause-tries during a goal reduction, it is reasonable to assume that the average clause-try execution time \bar{t}_{ct} is greater than the average remaining goal reduction execution time \bar{t}_r , that is, $0 < \kappa \leq 1$. In Figure 5.5 we show the relative execution time O_{ct} for $(0 < \kappa \leq 1)$ and for a set of cases where the fraction of redundant clause-tries is $F_{ct}^r = 0.8$ and also for several cases where $(0.5 \leq F_{ct}^r \leq 0.8)$.

For the case where $F_{ct}^r = 0.8$, the relative execution time of redundant clause-tries ranges from 80% for $\kappa \rightarrow 0$ to 40% for $\kappa = 1$. This is a significant amount which should be reduced in a special-purpose implementation. The relative execution time O_{ct} can be reduced by reducing the average execution time of a clause-try \bar{t}_{ct} . This is possible using improved compilation techniques, as described in Chapter 4.

Summary of Results

- We have defined a performance model that describes the relative execution time of redundant clause-trying during goal reduction, O_{ct} .
- In the selected benchmark programs, the average number of redundant clause-tries is very high, that is, $F_{ct}^r = 0.77$
- Depending on the relative average *weight* of clause-tries \bar{t}_{ct} , compared to the average execution time of the remaining part of goal reduction \bar{t}_r , and for a range of values $0 < \kappa = \frac{\bar{t}_r}{N_{ct}^r \bar{t}_{ct}} \leq 1$, the relative execution time of redundant clause-tries O_{ct} ranges from 40% to more than 80% of program execution time.
- Reducing the overhead of redundant clause-tries may be achieved using program compilation techniques that may significantly improve goal reduction performance. Therefore, rather than introducing special-purpose architectural support to reduce the execution time of clause-tries, we consider this optimization to be within the domain of compilation techniques.

5.5.3 Goal Suspension, Activation and Management

In [Fost87], the overhead of goal suspension and goal activation is labeled as one of the possible factors resulting in degraded FCP performance on a single processor. Goal suspension and activation is an implementation mechanism used to capture the asynchronous inter-goal communication characteristic of FCP. It is suspected to create a bottleneck because of the complexity of the suspension mechanism described in Chapter 4 and because of the frequency with which it occurs.

More generally, goal suspension and activation represents one aspect of the *dynamic* behavior of FCP goals. In addition, a goal may *create* new goals, *terminate* execution or *iterate*. We refer to this *dynamic* behavior of goals as *goal management*. In the following sections we define a performance model for the relative execution time of all goal management operations during a goal reduction. We then describe the program and system parameters followed by the model analysis and results.

5.5.3.1 Performance Model

Let \bar{t}_{ct} denote the average clause-try execution time and N_{ct}^r the average number of clause-tries per goal reduction. Let us divide the remaining goal reduction execution time into the average execution time of goal suspensions and the average execution time of the clause-body, \bar{t}_{body} . We can represent the average goal reduction execution time \bar{t}_{red} as follows:

$$\bar{t}_{red} = N_{ct}^r \bar{t}_{ct} + F_{susp} N_{var}^s \bar{t}_{susp} + (1 - F_{susp})(N_{act}^c \bar{t}_{com} + \bar{t}_{body}) \quad (5.10)$$

where F_{susp} denotes the frequency of goal suspensions, that is, the Average Suspension Rate (ASR), N_{var}^s the average number of variables a goal suspends on, \bar{t}_{susp} represents the average goal suspension execution time per suspension variable, N_{act}^c the average number of activated goals per goal-commit and \bar{t}_{com} the average goal activation time per activated goal.

To represent the average execution time of a clause-body, let us consider the following situation. For each goal that is reduced, its arguments had to be created, that is, *put* into the goal record. N_{sp} of the reduced goals were actually spawned and N_h of them terminated. Let \bar{t}_p denote the average *put* time for each argument of the created goal, \bar{t}_{sp} represents the average spawn time and \bar{t}_h represents the average halt time. The average clause-body execution time per goal reduction \bar{t}_{body} is represented as:

$$\bar{t}_{body} = N_{arg} \bar{t}_p + F_h \bar{t}_h + F_{sp} \bar{t}_{sp} \quad (5.11)$$

The complete expression for the average goal reduction time is now expressed as:

$$\bar{t}_{red} = N_{ct}^r \bar{t}_{ct} + F_{susp} N_{var}^s \bar{t}_{susp} + (1 - F_{susp})(N_{act}^c \bar{t}_{com} + N_{arg} \bar{t}_p + F_h \bar{t}_h + F_{sp} \bar{t}_{sp}) \quad (5.12)$$

The relative execution time of goal management during goal reduction is represented using the general expression described earlier. That is,

$$O_{gm} = \frac{1}{1 + \frac{N_{ct}^r \bar{t}_{ct} + N_{arg}(1 - F_{susp}) \bar{t}_p}{F_{susp} N_{var}^s \bar{t}_{susp} + (1 - F_{susp})(N_{act}^c \bar{t}_{com} + F_h \bar{t}_h + F_{sp} \bar{t}_{sp})}} \quad (5.13)$$

In the above expression for the relative execution time of goal management operations O_{gm} , all execution times \bar{t}_i are implementation dependent, whereas the rest of the parameters are program dependent parameters obtained using the *slogix* system. To simplify the expression we make the following assumptions:

- $F_h = F_{sp}$. The frequency of goal creation is equal to the frequency of goal termination. That is, we consider program completion when all the created goals terminate.
- $\bar{t}_h = \bar{t}_{sp} = t$. The execution time of a *halt* and *spawn* operation are both equal to t time units in a given implementation. Both operations manipulate the active goal queue and are of the same complexity.
- $\bar{t}_{susp} = \bar{t}_{com} = \mu t$. The execution time of a goal suspension per suspended variable is equal to the activation time per activated goal and both operations execute in μt units of time in a given implementation.

Let us further represent the average execution time of a clause-try as:

$$\bar{t}_{ct} = \varepsilon \times t \quad (5.14)$$

and the average execution time of a *put* argument as:

$$\bar{t}_p = \xi \times t \quad (5.15)$$

The relative goal management execution time is now simplified to the following expression:

$$O_{gm} = \frac{1}{1 + \frac{\varepsilon N_{ct}^r + N_{arg}(1 - F_{susp})\xi}{F_{susp} N_{var}^s \mu + (1 - F_{susp})(N_{act}^c \mu + 2F_h)}} \quad (5.16)$$

Therefore, 3 variable parameters are defined in the above expression: μ represents the execution time of suspend and commit relative to spawn and halt, ε denotes the ratio of average clause-try execution time versus the spawn or halt execution time and ξ denotes the ratio of average *put* time per goal argument versus spawn or halt execution time. In addition, the proposed performance model defines the following 6 system parameters:

1. F_h represents the number of goal creations per goal reduction.
2. F_{susp} is the goal suspension rate per goal reduction (ASR).
3. N_{ct}^r represents the average number of clause-tries per goal reduction.
4. N_{arg} denotes the average number of goal arguments in a goal.
5. N_{var}^s is the average number of suspension variables used for suspension.
6. N_{act}^c represents the average number of goals activated at goal commit.

We now present measurements obtained using *slogix*.

Goal:	FCP Benchmark Programs						
	Compiler	Sim.1	Sim.2	Debug	Solver	Distr.	Logix
N_{sp}	1890235	1133169	2501064	573869	156491	104799	477568
N_h	1887576	1131614	2499518	572005	155306	97184	474729
N_{susp}	887590	1892561	2513692	502659	52448	83315	306450
N_{act}	884945	1891022	2512160	500816	51277	75714	303625
N_{red}	7075380	3009280	7722376	1593286	409075	259283	1481900

Table 5.3: Goal Management Statistics

	FCP Benchmark Programs						
	Compiler	Sim.1	Sim.2	Debug	Solver	Distr.	Logix
F_h	0.27	0.38	0.32	0.36	0.38	0.4	0.32
F_{susp}	0.1	0.6	0.33	0.32	0.13	0.3	0.21
AGM	0.8	2	1.29	1.3	1.3	1.2	1.1

Table 5.4: Goal Management Parameters

5.5.3.2 Performance Model Parameters

In table 5.3 we show the number of goals that are created and terminated during the execution of the benchmark programs, as well as the total number of performed goal reductions. Also shown are the number of goal suspensions and activations.

Frequency of Goal Creations, F_h

In table 5.4 we show the number of goal creations per goal reduction F_h . It ranges from 0.27 to 0.4, with the average value 0.35. The reciprocal value of F_h denotes the number of goal reductions per created goal. On the average, a goal performs between 2 to 4 reductions before it terminates, with the mean value being 2.9 reductions. In terms of goal reductions, a *typical* goal in the FCP programs is a *light weight* or *short-lived* computation, that iterates on the average 3 reductions prior to termination. There is not a simple correlation between a goal reduction and actual execution time, which is implementation dependent.

Average Goal Suspension Rate, F_{susp}

The goal suspension rate F_{susp} , varies more significantly amongst the selected programs. The *Compiler* performs on the average 8 reductions prior to a goal suspension and the *Solver* performs less than 2 reductions prior to suspension. The average value is $F_{susp} = 0.26$ suspensions per reduction.

It is interesting to note that the benchmarks used in [Tick88] recorded a significantly smaller number for F_{susp} , that is, between 0 and 0.09. Whereas this value is closer to the characteristic behavior of the *Compiler* program, on the average the benchmarks that we use exhibit almost 3 times the maximum ASR value found in Tick's benchmarks. It is our observation that larger programs in FCP exhibit a more complex behavior than the programs used in [Tick88], which were also optimized for performance.

If we consider both the rate of goal suspension and activation as a single parameter (approximately $2 \times F_{susp}$), then on the average a goal suspends or activates other goals every 1.5 reductions. In the selected programs, the maximum rate corresponds to the *Simulator1* program, which suspends or activates goals 1.25 times per goal reduction. This implies that goals suspend more than once before being reduced. Such a situation occurs when a goal that is suspended on several variables, receives data from just one variable. The goal is nevertheless activated since there may exist a matching clause that results in goal reduction.

In Table 5.4 we also show the average goal management activity per goal reduction, AGM, for all of the selected programs. It is interesting to see a surprisingly similar behavior. For all of the programs except *Simulator2*, the value for AGM is close to 1 and for *Simulator2* it is equal to 2. That is, on the average, a goal reduction will perform approximately one goal management operation per reduction. We say "surprisingly", since the selected programs represent a variety of program applications written by different programmers. Nevertheless, the goal management characteristics are quite similar, thus indicating a consistent behavior typical of large FCP programs and of the selected system's workload.

Frequency of Clause-Tries, N_{ct}^r

In Table 5.2 used during the analysis of the redundant clause-try relative execution time, one of the table entries was the average number of reductions per clause-try, CR/CT . This is the reciprocal value of N_{ct}^r . We note that the

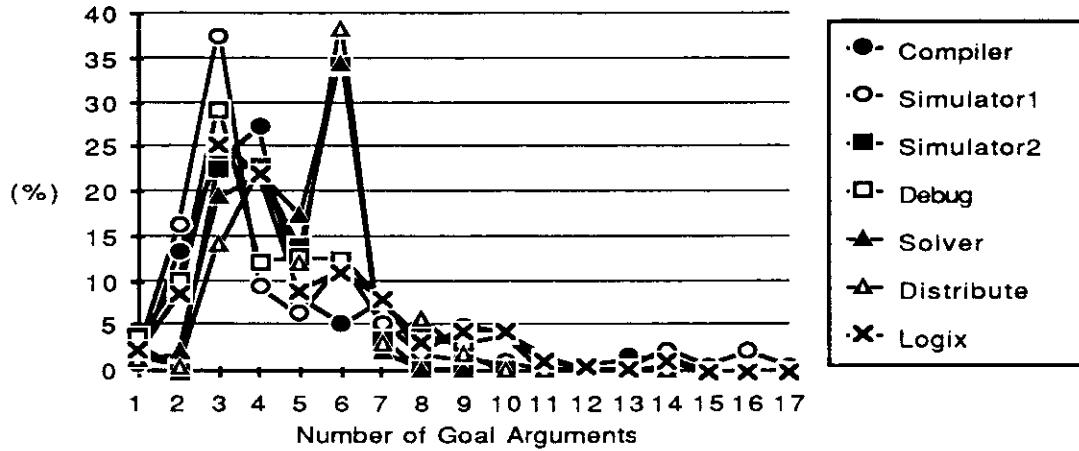


Figure 5.6: Distribution of the Number of Goal Arguments

number of clause-tries per goal reduction varies from 3 for the *Compiler* program to 11 in the case of the *Solver* program. The average value is $N_{ct}^r = 5.8$.

Number of Goal Arguments, N_{arg}

In Figure 5.6 we show the distribution of the number of goal arguments found in the selected FCP benchmarks. The goal size indicates the number of program arguments and does not include the program counter (or perhaps some other system arguments that may be stored as part of a goal).

In Table 5.5 we also show the total number of allocated arguments, the maximum number of arguments found in a single goal and the average number of arguments per created goal. From the distribution of the number of goal arguments shown in Figure 5.6 we see that most of the goals have less than 7 arguments. Whereas the maximum number of goal arguments is between 14 and 17, these occur very rarely. The average size is quite similar in all of the selected programs, and ranges from 4.5 to 5.1. The overall mean value for the number of goal arguments is $N_{arg} = 4.7$.

Goal Suspension Variables, N_{var}^s

In Figure 5.7 we show the distribution of the number of variables a goal suspends on during program execution. The results reconfirm what was assumed about flat committed choice languages, namely that they suspend mostly on very

FCP Benchmark Programs							
	Compiler	Sim1	Sim2	Debug	Solver	Distr.	Logix
Tot. (10^6)	39.70	22.94	51.43	9.77	2.15	1.75	8.67
GT (10^6)	7.96	5.90	10.23	2.09	0.46	0.34	1.78
Ave.	4.9	3.9	4.6	4.7	4.7	5.1	4.9
Max.	14	17	14	14	14	14	14

Table 5.5: Goal Arguments: Maximum, Average

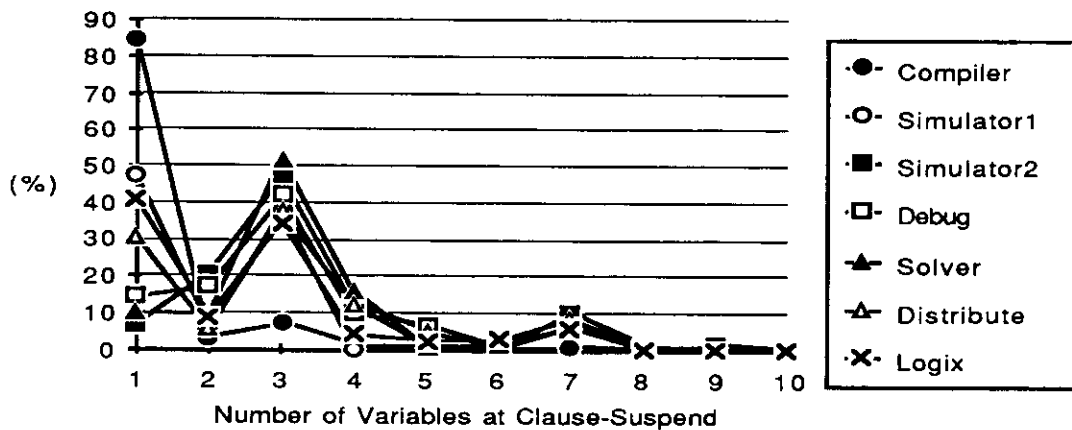


Figure 5.7: Distribution of the Number of Suspension Variables

few variables. However, whereas it was assumed that most suspensions are on one or two variables and less frequently three or more, only the *Compiler* program exhibited this type of expected characteristic. The rest of the programs showed a higher distribution for suspensions on 3 variables rather than 1 or 2.

This type of *unexpected* goal suspension characteristic can be explained as follows. First, the behavior of goal suspension in large applications of flat committed choice languages was never precisely evaluated. Second, the observation that suspension occurs usually for 1, 2 and less frequently 3 variables was usually made on small applications that do not exhibit the program complexity typical of large applications. And third, it is readily assumed that more advanced compilation techniques can eliminate redundant suspensions on more variables. Whereas this may be true, such compilation techniques are still a matter of research and are not yet available.

In Table 5.6 we show the total number of variables that goals suspend on

FCP Benchmark Programs							
	Compiler	Sim.1	Sim.2	Debug	Solver	Distr.	Logix
Total	1228145	4340234	8281821	1590037	170657	235218	809585
N_{susp}	886526	1891370	2512452	501468	51355	82122	305386
Ave.	1.4	2.3	3.3	3.1	3.3	2.8	2.5
Max.	8	9	12	14	14	10	15
(1-4)	97	91.9	87.2	84	87.7	86.9	88.8

Table 5.6: Suspending Variables at Suspend

during program execution as well as the total number of goal suspensions. Also shown are the maximum and average number of variables a goal suspends on. Note that the maximum value ranges from 8 variables for the *Compiler* to 15 variables for the *Logix* operating system. Also, the average values are higher than was expected. It ranges from 1.4 for the *Compiler* program to 3.3 for the *Solver* program.

Therefore, based on the benchmarked FCP programs, one can see that goal suspension occurs mainly on few variables. In Table 5.6 we show that the percentage of all goal suspensions that occurred with 4 or less variables. This ranges from 84% for the *Debugger* to 97% for the *Compiler*. However, the average number of variables a goal suspends on is higher than expected. Closer to 3 variables than 1. The mean value for the number of suspension variables is $N_{var}^s = 2.6$.

Goal Activation Rate, N_{act}^c

In Figure 5.8 we show the distribution of the number of activated goals at clause-commit. One should note that in most cases, a clause-commit will not activate any goal. That is, either there were no new assignments made during the clause-head unification and guard evaluation, or, if there were assignments made, there were no goals suspended waiting for these values. Only in the case of the *Simulator1* program were as few as 50% of the commits empty, whereas in the other programs as many as 88% and more clause-commits did not activate any goals.

From Table 5.7 we see that the average number of goals activated per non-empty clause-commit is generally slightly over 1. However, the maximum number

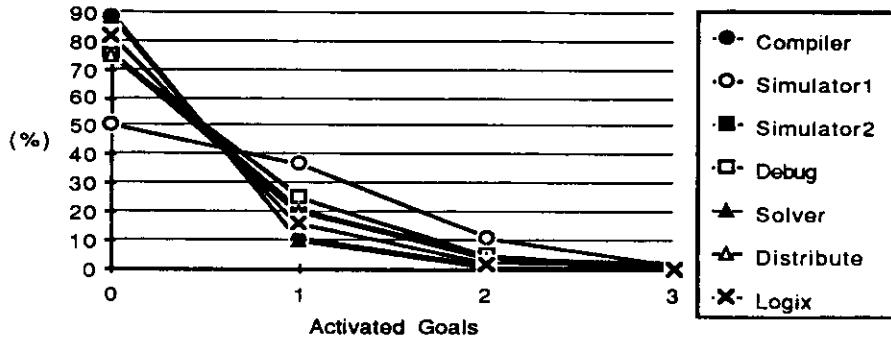


Figure 5.8: Distribution of Goals Activated at Clause-Commit

Goals	FCP Benchmark Programs						
	Compiler	Sim.1	Sim.2	Debug	Solver	Distr.	Logix
N_{act}^c	884945	1891022	2512160	500816	51277	75714	858276
Ave.	1.1	1.3	1.3	1.2	1.1	1.2	1.2
Max.	192	12	23	10	4	20	142

Table 5.7: Activated Goals at Clause-Commit

of goals activated may be unexpectedly high. In the *Logix* system benchmark the maximum number of goals activated at any time was 142 and in the *Compiler* as many as 192. This situation corresponds to data broadcasting where many goals are waiting for the same data element.

Performance Model Variables

The three variables defined in the goal management performance model represent implementation dependent ratios of goal management operation execution times and goal reduction operations of a clause-try, \bar{t}_{ct} , or argument creation \bar{t}_p . In general, the goal management operations: suspend, commit, spawn and halt are high-level operations that are more complex and time consuming when executed in a sequential, general-purpose environment. We now consider the range of variable values that are reasonable for performance analysis.

- μ : The suspend and commit operations are more complex than spawn and halt. Therefore, the condition $\mu > 1$ is assumed. In the performance model

analysis we consider the range of values $1 \leq \mu \leq 20$. This ratio depends on such implementation issues such as the type of machine instructions.

- ε : The average execution time of a clause-try is considered to be more complex than the spawn and halt operations. During a clause-try, each argument in the clause is matched with the arguments of a calling goal. The arguments are previously *dereferenced*. The matching process called unification may require variable *trailing*. These aspects of a clause-try are modeled later in this chapter. We consider that the clause-try time is greater than the time to perform a spawn or halt operation. Therefore, we consider the case where $\varepsilon \geq 1$, and a range: $1 \leq \varepsilon \leq 5$.
- ξ : We assume that the halt and spawn operations are more complex than the average execution time of a single argument *put* operation. We therefore consider $0 < \xi \leq 1$.

5.5.3.3 Performance Model Analysis

In Figure 5.9 we show the relative execution time of goal management operations O_{gm} during goal reduction. The average values for the performance model system parameters are: $F_{susp} = 0.26$, $F_h = 0.35$, $N_{arg} = 4.7$, $N_{ct}^r = 5.8$, $N_{var}^s = 0.26$ and $N_{act}^c = 0.2$. Four different cases for the relative execution time are shown in Figure 5.9, for values $\xi \in (0.25, 0.5, 0.75, 1)$ (shown clockwise from top left). In each diagram $1 \leq \varepsilon \leq 5$ is used as the function parameter.

Let us consider the execution of FCP in a general-purpose, sequential environment for the System's Development Workload defined by the set of system parameter values. Let us further assume that the work-point is defined to be in the middle of the defined performance variables range of values. That is, for the workpoint defined by $\xi = 0.5$, $\varepsilon = 3$ and $\mu = 10$, we note that the relative execution time is 35%, that is, $O_{gm} = 0.35$.

It is assumed in the presented diagrams that the compilation technique is the one used in the abstract machine which performs clause-selection in textual order. Thus, the system parameter that denotes the number of clause-tries per reduction is $N_{ct}^r = 5.8$. Using a better compilation technique, the clause-try execution time is proportional to the number of arguments of the clause-try rather than the product of the number of arguments and the number of clauses.

In Figure 5.10 we show the relative execution time of goal management exe-

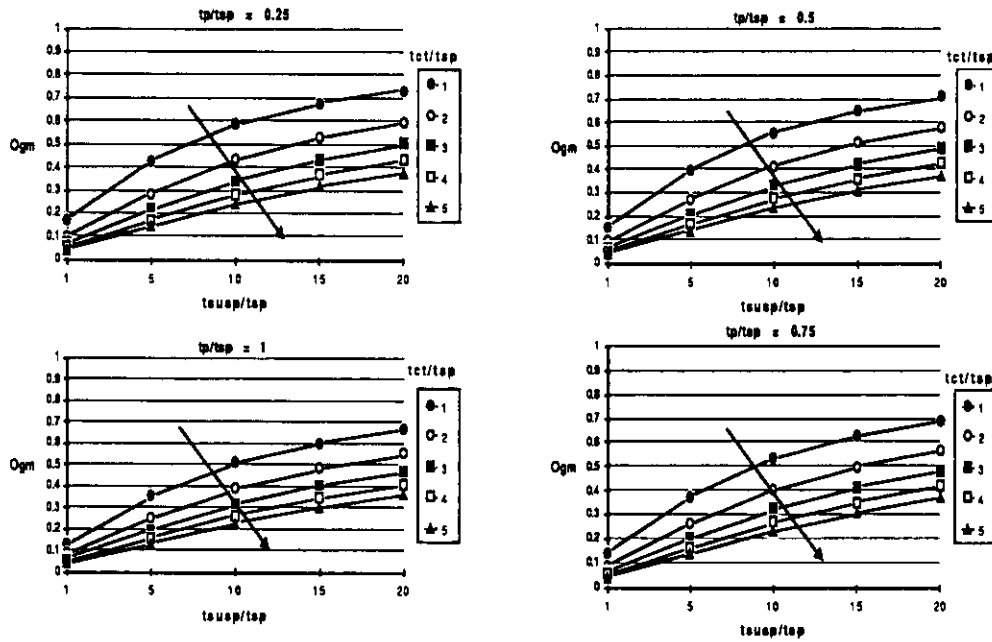


Figure 5.9: Relative Execution Time O_{gm} , $N_{ct}^r = 5.8$

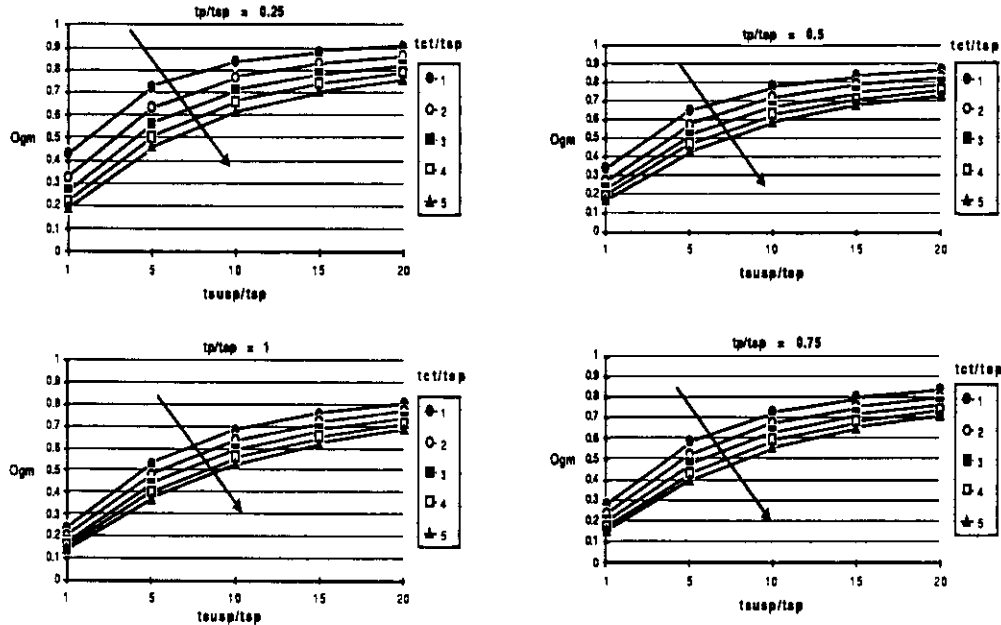


Figure 5.10: Relative Execution Time O_{gm} , $N_{ct}^r = 1$

cution, but this time for the program parameter $N_{ct}^r = 1$. We notice a significant increase in the relative execution time, for the same set of system parameter values and for the workpoint defined by $(\xi = 0.5, \varepsilon = 3, \mu = 10)$. In this case, $O_{gm} = 0.7$.

5.5.3.4 Goal Management Statistics Summary

We now summarize the above presented goal management statistics for the selected large FCP programs.

- A performance model for the analysis of the relative execution time of goal management operations, O_{gm} is defined. It consists of 6 program parameters and 3 implementation dependent variables.
- The relative goal management execution time O_{gm} , for a reasonably selected working point characterizing a general-purpose processor implementation,

and for the specific system workload defined by the system parameter values, is very high, 35% .

- Using decision-tree compilation the overhead of clause-try is reduced and the relative goal management execution time is further pronounced.
- Therefore, architectural support for goal management operations is a high-priority in the design of a special-purpose processor architecture for the execution of FCP.

In addition to the results of the performance model analysis, a number of additional observations can be made regarding goal management operations in general.

- The execution of large FCP programs such as compilers, simulators or development systems (like the Logix operating system) result in the frequent creation of concurrent, cooperating and communicating goals.
- Goals are *light weight* computations relative to the number of goal reductions. That is, the average life time of a goal is between 2 or 3 goal reductions.
- Due to the asynchronous nature of inter-goal communication, FCP goals frequently *suspend* execution waiting for data to be communicated by another goal. The average goal suspension rate (ASR) in the selected programs is 0.29. Goal suspension and activation occurred as often as 1.5 times per reduction. A note should be made that the ASR observed in the selected workload almost 3 times larger than the suspension rate reported in [Tick88].
- In almost 90% of the cases, a goal suspends on 4 or less variables. However, the average number of variables a goal suspends on is $N_{var}^s = 2.6$.
- Suspended goals are activated only at clause-commit. Between 50-88% of all clause-commits do not activate goals. In most cases, however, only a single goal is placed onto the active goal queue. The average number of activated goal per goal-commit is $N_{act}^c = 0.2$.
- An FCP goal frequently performs goal management operations. The average goal management activity, AGM, is 1.2 goal management operations per goal reduction.

5.5.4 Dereferencing

A goal argument is denoted as a reference to a program data structure or as a reference to another reference. It is for this reason that an argument reference is *dereferenced* prior to matching with another program data structure. Argument dereferencing consists of following a chain of reference pointers until a non reference is encountered. It is performed not only during goal-head unification for matching goal arguments, but also prior to the assignment of a value to a variable. In general, dereferencing is a frequent operation in FCP.

5.5.4.1 Performance Model

Let N_d^{ct} denote the average number of dereference calls per clause-try, \bar{l}_d the average length of a dereference chain and \bar{t}_d the average execution time of a dereference operation per unit length. The average goal reduction time \bar{t}_{red} is then represented as follows:

$$\bar{t}_{red} = N_{ct}^r \bar{t}_{ct}^d + N_{ct}^r N_d^{ct} \bar{l}_d \bar{t}_d + \bar{t}_r \quad (5.17)$$

where \bar{t}_{ct}^d is the average clause-try execution time without dereferencing, and \bar{t}_r is the remaining goal reduction time. As in the goal management performance model N_{ct}^r denotes the average number of clause-tries per goal reduction. The relative execution time of argument dereferencing is now represented as:

$$O_d = \frac{1}{1 + \frac{\frac{\bar{t}_{ct}^d}{\bar{t}_d} + \frac{\bar{t}_r}{N_d^{ct} \bar{t}_d}}{N_d^{ct} \bar{t}_d}} \quad (5.18)$$

Let us denote the ratios of clause-try execution time and the remaining goal reduction time (both without dereferencing) versus the execution time of a dereference of a unit length as:

$$\frac{\bar{t}_{ct}^d}{\bar{t}_d} = \beta \quad (5.19)$$

$$\frac{\bar{t}_r}{\bar{t}_d} = \gamma \quad (5.20)$$

The relative execution time of argument dereferencing is now expressed as:

$$O_d = \frac{1}{1 + \frac{\beta + \frac{\gamma}{N_d^{ct}}}{N_d^{ct} \bar{t}_d}} \quad (5.21)$$

The parameters β and γ are performance model variables whereas N_d^{ct} , \bar{l}_d and N_{ct}^r are system parameters. We now discuss the performance model parameters.

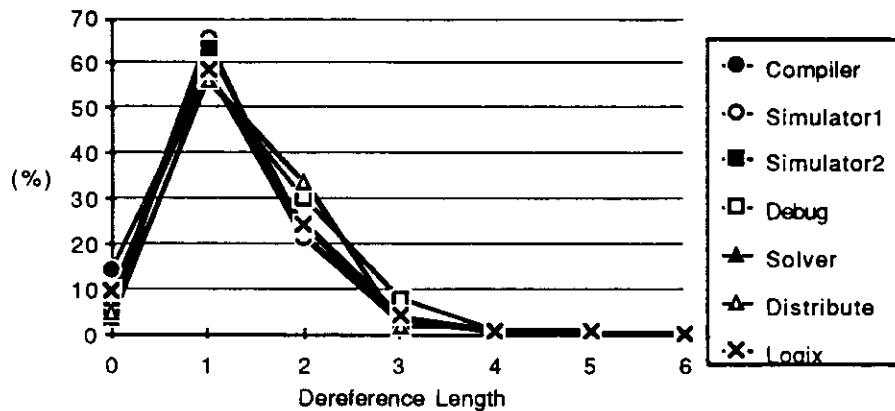


Figure 5.11: Distribution of Dereference Length

	FCP Benchmark Programs						
	Compiler	Sim.1	Sim.2	Debug	Solver	Distr.	Logix
D (10^6)	86.15	74.50	174.25	67.94	13.96	8.49	25.62
Len. (10^6)	115.42	183.74	481.52	115.29	28.47	23.25	47.49
$\bar{l}_d = L/D$	1.34	2.46	2.76	1.69	2.03	2.73	1.85
CT (10^6)	20.14	18.94	48.99	7.46	4.52	2.59	5.85
$N_d^{ct} = D/CT$	4.28	3.93	3.56	9.09	3.08	3.27	4.38

Table 5.8: Dereferencing Statistics

5.5.4.2 Performance Model Parameters

System Parameters

In Figure 5.11 we show the distribution of the dereference length in the FCP benchmark programs. We see that most of the dereference chains are less than 3. In Table 5.8 we show the total number of dereference calls (D) made in the FCP programs, the total dereference length (Len) and the average dereference length per dereference call \bar{l}_d . We also show the average number of dereference calls per clause-try, $N_d^{ct} = 4.5$. The overall mean value of the dereference length is $\bar{l}_d = 2.1$.

Even though the average dereference length is between 1 and 3 in most programs, one should note that there are also dereference chains of length 400 and more. This is not so surprising since it is quite easy to form large chains of references by repeated unification of logical variables. This case, for example,

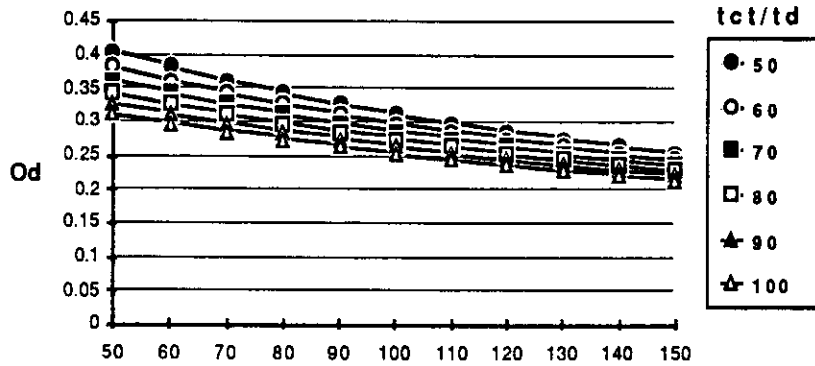


Figure 5.12: Relative Execution Time of Argument Dereferencing

occurs during the use of the *short circuit* technique used for detecting program termination.

Performance Model Variables

The two implementation dependent performance model variables β and γ denote the relative execution times of clause-try and remaining goal reduction execution time (both without argument dereferencing) versus the execution time of a dereference operation of unit length, \bar{t}_d . Since \bar{t}_d consists of a memory read plus operations such as tag extraction and comparison, one can safely assume that $\beta \gg 1$ and $\gamma \gg 1$. We now discuss the performance model analysis for a specific range of variable values.

5.5.4.3 Performance Model Analysis

In Figure 5.12 we show the relative execution time of argument dereferencing, for the range of values ($50 < \beta < 100$) and ($100 < \gamma < 150$).

For values of $\gamma = 100$ and $\beta = 50$ the relative execution time of argument dereferencing is as high as 16%. This value decreases as both γ and β increase. It is assumed here that the program compilation did not include the possible optimizations of the decision tree type. That is, the number of argument dereferencing is proportional to the product of the number of arguments and the number of clauses in a procedure. Therefore, the relative execution time of dereferencing

can be reduced using decision tree compilation.

5.5.4.4 Summary and Conclusion

- A performance model for the analysis of the relative execution time of dereferencing during goal reduction, O_d is defined. It consists of 3 program parameters and two implementation dependent variables.
- The relative execution time of argument dereferencing for the specific system workload ranges from 10% to 16% for a range of system variable values ($50 \leq \beta \leq 100$) and ($100 \leq \gamma \leq 150$).
- Using improved compilation techniques, the number of argument dereferencing calls can be reduced, and thus the relative execution time of dereferencing can also be reduced.

In general, the following characteristics have been observed regarding dereferencing in the selected FCP programs:

- Dereferencing is a frequent operation performed on the average $N_d^{ct} = 4.5$ times per clause-try.
- The average length of a dereference operation is $\bar{l}_d = 2.1$.

5.5.5 Clause-Try Trailing

As described in Section 4.1 and shown in Figure 4.2 a clause-try may result in the assignment of values to logical variables. If a clause-try succeeds, the bindings become visible to the programming environment and the goal reduction commits. However, if a clause-try fails or suspends, the assignments to memory must be undone so that a new clause-try may begin with the same memory state that preceded the clause-try. It is for this reason that the assignments to memory are *trailed* during a clause-try.

Trailing consists of storing the address and previous value of a trailed logical variable. For this purpose, the *Trail Stack* is used in the sequential abstract machine for FCP. Prior to assigning a value to the logical variable, the variable address and old value are pushed into the stack. In case of clause-try failure or suspension, the entries are popped from the trail, and memory is restored to the state preceding the last clause-try.

5.5.5.1 Performance Model

Let \bar{t}_t denote the average execution time of clause-trailing during a clause-try. The average goal reduction time \bar{t}_{red} is then represented as follows:

$$\bar{t}_{red} = N_{ct}^r \bar{t}_{ct}^t + N_{ct}^r \bar{t}_t + \bar{t}_r \quad (5.22)$$

where \bar{t}_{ct}^t is the average clause-try execution time without trailing and \bar{t}_r is the remaining goal reduction time. Since the execution time of trailing a logical variable depends on whether the clause-try committed, suspended or failed, we represent the average trailing time per clause-try as follows:

$$\bar{t}_t = \bar{N}_t^c \bar{t}_t^c + \bar{N}_t^s \bar{t}_t^s + \bar{N}_t^f \bar{t}_t^f \quad (5.23)$$

where \bar{N}_t^c , \bar{N}_t^s , and \bar{N}_t^f denote the average number of variables trailed per clause-commit, clause-suspend and clause-failure respectively. Similarly, \bar{t}_t^c , \bar{t}_t^s , and \bar{t}_t^f denote the execution time of clause-trailing per unit element during clause-commit, clause-suspend and clause-failure respectively. The difference in the execution times is that a trailed element during a clause-try that suspends or fails also has to include the time to restore the old variable value. In case of clause-commit, the execution time consists of only trailing.

Therefore, the average execution time of goal reduction is represented as:

$$\bar{t}_{red} = N_{ct}^r \bar{t}_{ct}^t + N_{ct}^r (\bar{N}_t^c \bar{t}_t^c + \bar{N}_t^s \bar{t}_t^s + \bar{N}_t^f \bar{t}_t^f) + \bar{t}_r \quad (5.24)$$

Since the execution time of trailing during a clause-suspend is the same as during clause-failure, that is, $\bar{t}_t^s = \bar{t}_t^f$, the relative execution time for clause-try trailing is now represented as:

$$O_t = \frac{1}{1 + \frac{N_{ct}^r \bar{t}_{ct}^t + \bar{t}_r}{N_{ct}^r [\bar{N}_t^c \bar{t}_t^c + \bar{t}_t^s (\bar{N}_t^s + \bar{N}_t^f)]}} \quad (5.25)$$

This expression can be further simplified if we assume that the execution time of clause-suspend is twice the execution time of clause-commit, that is, $\bar{t}_t^s = 2 \times \bar{t}_t^c$:

$$O_t = \frac{1}{1 + \frac{\bar{t}_{ct}^t + \bar{t}_r}{\bar{N}_t^c + 2 \times (\bar{N}_t^s + \bar{N}_t^f)}} \quad (5.26)$$

Let us denote the ratios of clause-try execution time without trailing and the remaining goal reduction time, versus the execution time of a trail for a single variable during a clause-try that commits, as follows:

$$\frac{\bar{t}_{ct}^t}{\bar{t}_t^c} = \tau \quad (5.27)$$

$$\frac{\bar{t}_r}{\bar{t}_t^c} = \rho \quad (5.28)$$

The relative execution time of clause-try trailing is now expressed as:

$$O_t = \frac{1}{1 + \frac{\tau + \frac{\rho}{N_{ct}^r}}{N_t^c + 2(N_t^s + N_t^f)}} \quad (5.29)$$

The parameters τ and ρ are implementation dependent parameters that are used as variables in the proposed model whereas \bar{N}_t^c , \bar{N}_t^s , \bar{N}_t^f and N_{ct}^r are system parameters described in the following subsection.

5.5.5.2 Performance Model Parameters

System Parameters

In Table 5.9 we show the average number of trailed variables stored in the trail stack during a clause-try that commits, suspends and fails, for the selected FCP programs. We also show the average *length* of trailed elements during those clause-tries that committed: $\bar{l}_t^c = 5$, suspended: $\bar{l}_t^s = 1.4$ and failed: $\bar{l}_t^f = 0.5$. Therefore, one can note the following. Most of the trailing occurs during the clause-tries that succeed. In fact, an order of magnitude more than during the clause-tries that fail. In other words, most clause-tries fail before they require much trailing.

In Figure 5.13 we show the distribution of the trail size at clause-commit. Even though the behavior of the distribution is not the same for each of the selected FCP programs, in most cases the trail size is less than 5 or 6 entries. In Table 5.9 we show the average and maximum size of the trail, as well as the total number of trailed entries at clause-commit clause-suspend and clause-failure. Whereas the average size is quite small for each program, the maximum number of trailed entries may be high. In the case of the *Distribute* program as many as 163 entries were trailed. This, however, occurred very infrequently.

FCP Benchmark Programs							
	Compiler	Sim.1	Sim.2	Debug	Solver	Distr.	Logix
CR (10^6)	7.07	3.00	7.72	1.59	0.40	0.25	1.48
$L_c(10^6)$	23.78	20.35	36.31	9.74	2.32	1.41	6.76
Max	36	47	22	35	54	163	87
\bar{l}_t^c	3.4	6.8	4.7	6.1	5.6	5.5	4.6
\bar{N}_t^c	1.2	1.1	0.7	1.3	0.5	0.5	1.2
CS (10^6)	2.41	12.35	16.33	2.92	0.31	0.54	1.52
$L_s(10^6)$	2.81	16.46	29.21	5.05	0.50	1.01	2.41
Max	12	16	18	21	21	26	92
\bar{l}_t^s	1.1	1.1	1.2	1.6	1.6	1.7	1.4
\bar{N}_t^s	0.1	0.9	0.6	0.7	0.1	0.4	0.4
CF (10^6)	10.65	3.58	24.93	2.94	3.79	1.79	2.84
$L_f(10^6)$	2.76	1.38	13.76	2.25	3.46	1.62	1.81
Max	38	10	12	20	46	18	81
\bar{l}_t^f	0.2	0.4	0.5	0.4	0.9	0.8	0.5
\bar{N}_t^f	0.1	0.1	0.3	0.3	0.8	0.6	0.3

Table 5.9: Trailing at Clause-Commit

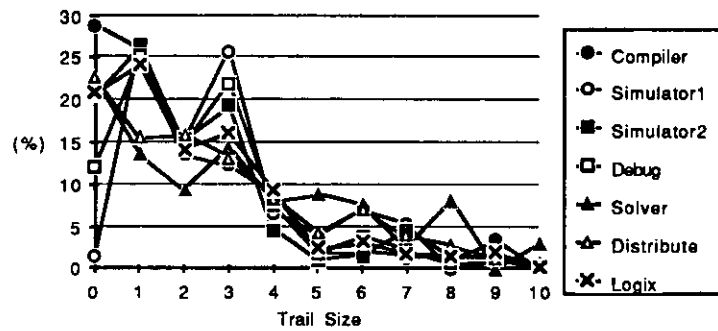


Figure 5.13: Distribution of Trail Size at Clause-Commit

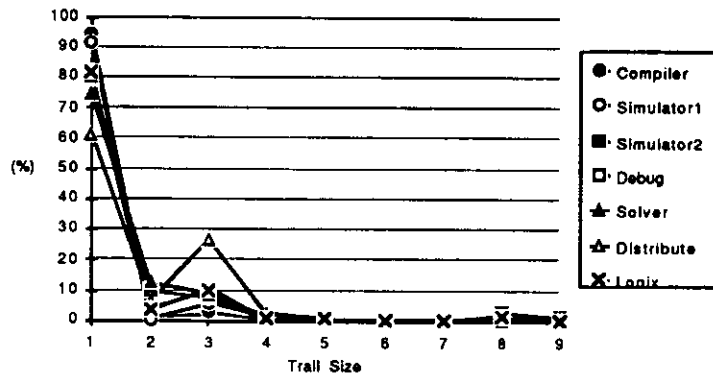


Figure 5.14: Distribution of Trail Size at Clause-Suspend

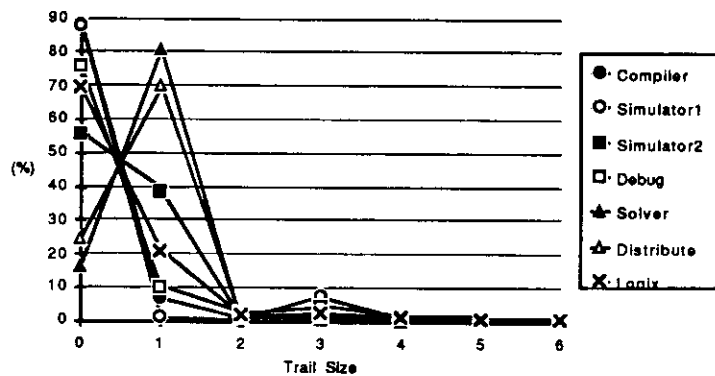


Figure 5.15: Trailing at Clause-Failure

In Figure 5.14 we show the distribution of the trail size at clause-suspend. It is quite similar for all the selected FCP programs. In most cases, the trail size is less than 4 entries. In Table 5.9 we give the total number of clause-suspension, the total number of trailed entries, and the maximum and average size of the trail. The average size of the trail at clause-suspension is 1.4. This is smaller than the average trail size at clause-commit. Also, the maximum number of trailed entries is generally smaller, except in the case of the *Logix* benchmark.

In Figure 5.15 we show the distribution of the trail size at clause-failure. In most cases it is less than 2 entries. In Table 5.9 we can see that the average trail size is less than 1 in all of the benchmarked programs. However, the maximum number of trailed entries is also high, ranging from 10 for the *Simulator1* program to 81 in the case of *Logix*.

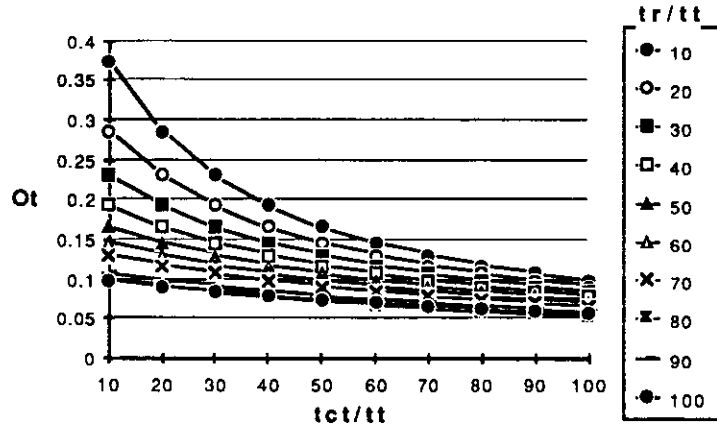


Figure 5.16: Relative Execution Time of Clause Trailing, O_t

System Variables

The implementation of variable trailing in a general-purpose environment consists of reading the value of a variable location and storing both the value and address pair into the trail stack. If we assume that the trail stack is located in memory, trailing a single variable consists of three memory accesses (1 to read variable value and 2 to store the variable address and value) and incrementing the trail stack pointer. Therefore, compared to the operations of a clause-try and the spawning of a clause-body, the following range of values for the system variables is reasonable: $\tau \gg 1$ and $\rho \gg 1$.

5.5.5.3 Performance Model Analysis

In Figure 5.16 we show the relative execution time of clause trailing for the range of variable values $10 \leq \tau \leq 100$ and $10 \leq \rho \leq 100$. In a general-purpose implementation it is expected that both τ and ρ will be large, thus resulting in a low relative execution time of clause trailing. However, additional special-purpose support may change this situation by either reducing the clause-try time (compilation techniques using decision tree analysis) or the goal management execution time.

For a range of values $10 \leq \tau \leq 60$ and $10 \leq \rho \leq 60$, the relative execution time of clause trailing is $5\% \leq O_t < 35\%$.

5.5.5.4 Summary and Conclusion

- A performance model for the evaluation of the relative execution time of variable trailing during a clause-try, O_t , is defined. It contains 4 program parameters and 2 implementation dependent variables.
- For the selected range of implementation dependent parameters, and for the specific program parameters, the relative execution time of clause-trailing was $5\% \leq O_t < 35\%$.

In general, the following observations regarding trailing were made:

- The average length of trailed variables per clause-try that commits is $\bar{l}_t^c = 5$, suspends is $\bar{l}_t^s = 1.4$, and fails $\bar{l}_t^f = 0.5$.
- The maximum number of trailed variables was in several cases as high as 160.

5.6 A General Goal Reduction Performance Model

In the previous sections we presented performance models that characterize the relative execution time of special purpose functions during program execution. We now combine these models into a general performance model. The advantage is the ability to capture the inter-dependency of the individual models. That is, it allows for changes in specific parameter values to be reflected in the general performance model.

The individual performance models characterize the average goal reduction execution time in terms of the specific functions that were considered as potential implementation bottlenecks. As a result, the average execution time of a clause-try was divided into the average dereferencing and trailing execution times. The remaining average execution time of a clause-try can be expressed as a function of the number of arguments of a clause-try, N_{arg} , and the average execution time per clause argument. Since the operations of a clause-try are commonly referred to as *get* operations, let \bar{t}_g denote the average execution time of a clause-try operation per clause argument. The average execution time of a clause-try is then expressed as follows:

$$\bar{t}_{ct} = N_{arg}\bar{t}_g + N_d^{ct}l_d\bar{t}_d + [\bar{N}_t^c + 2(\bar{N}_t^s + \bar{N}_t^f)]\bar{t}_t^c \quad (5.30)$$

$$\begin{aligned}
\bar{t}_{red} = & N_{ct}^r \times && \% \text{ clause - try per reduction} \\
& \{N_{arg} \bar{t}_g + && \% \text{ average get time} \\
& + N_d^{ct} \bar{l}_d \bar{t}_d + && \% \text{ dereferencing} \\
& + [\bar{N}_t^c + 2(\bar{N}_t^s + \bar{N}_t^f)] \bar{t}_t^c \} + && \% \text{ variable trailing} \\
& + F_{susp} N_{var}^s \bar{t}_{susp} + && \% \text{ goal suspension} \\
& + (1 - F_{susp}) \times && \% \text{ goal reduction} \\
& (N_{act}^c \bar{t}_{com} + && \% \text{ goal activation} \\
& + N_{arg} \bar{t}_p + && \% \text{ argument creation} \\
& + F_{sp} \bar{t}_{sp} + && \% \text{ goal creation} \\
& + F_h \bar{t}_h) && \% \text{ goal termination}
\end{aligned} \tag{5.31}$$

Figure 5.17: Goal Reduction Model

where the average execution time of a *get* operation does not include the time for dereferencing arguments and trailing variable assignments.

If we now combine the performance models for goal management with the above expression for the average clause-try execution time, the average goal reduction execution time, \bar{t}_{red} is represented in Figure 5.17.

System Parameters

The above defined performance model defines the following system parameters:

- N_{ct}^r . Average number of clause tries per goal reduction.
- N_{arg} . Average number of arguments per clause-try.
- N_d^{ct} . Average number of dereference calls per clause-try.
- \bar{l}_d . Average dereference length per dereference call.
- $\bar{N}_t^c, \bar{N}_t^s, \bar{N}_t^f$. Average number of variables trailed per clause commit, suspend and fail.
- N_{var}^s . Average number of suspension variables per goal suspension.
- N_{act}^c . Average number of goal activations per goal reduction.

- F_{susp} . Average number of goal suspensions per goal reduction.
- F_{sp} . Average number of goal creations per goal reduction.
- F_h . Average number of goal terminations per goal reduction.

The system parameters have been empirically characterized using the *slogix* system. The parameter values are specific for a workload that is used in a system's development environment. Similar characterization could be obtained for various system workloads and used in same the performance model. Moreover, various system workloads could be *synthesized* and used in the defined goal reduction performance model.

Implementation Dependent Parameters

The following implementation dependent parameters are part of the general performance model:

- \bar{t}_{susp} . The execution time of goal suspension per variable.
- \bar{t}_{com} . The execution time of goal activation per goal.
- \bar{t}_h . The execution time of goal termination.
- \bar{t}_{sp} . The execution time of goal spawning.
- \bar{t}_d . The execution time of a dereference of unit length.
- \bar{t}_t^c . The execution time of a single variable trailing during a clause-try that commits.

Two more parameters are defined that depend on machine implementation, compiler technology and program characteristics. In other words they are *architecture* dependent. These are:

- \bar{t}_g . The average execution time of a single argument matching during a clause-try, or *get* operation.
- \bar{t}_p . The average execution time of a single argument *put* during goal spawning.

General Performance Model Analysis

If we replace the system parameters that characterize the specific workload into the performance model expression, the result is a linear expression of the implementation dependent parameters. That is,

$$\bar{t}_{red} = \mathcal{F}(\bar{t}_{susp}, \bar{t}_{com}, \bar{t}_h, \bar{t}_{sp}, \bar{t}_d, \bar{t}_i^c, \bar{t}_g, \bar{t}_p) \quad (5.32)$$

Similarly, the performance model variables previously defined for the analysis of the specific goal reduction functions, also depend on the same implementation parameters. If for example, one were to change the implementation and reduce the average goal suspension execution time \bar{t}_{susp} , this would directly result in the reduced value of variable $\mu = \frac{\bar{t}_{susp}}{\bar{t}_h}$ used to evaluate the relative execution time of goal management shown in equation 5.16. The variables ξ and ε are not affected by the change. The result is a reduction in the relative execution time O_{gm} .

As a result of reducing \bar{t}_{susp} , the following system variables are also affected: γ , which is used to evaluate the relative execution time of argument dereferencing, O_d shown in equation 5.21, and ρ which is used to evaluate the relative execution time of variable trailing, O_t shown in equation 5.29. Therefore, changes in one implementation parameter are reflected in the overall distribution of the relative execution times of specific functions.

To perform a detailed *sensitivity* analysis of the general performance model relative to each implementation parameter is beyond the scope of this thesis; especially since it would require that a cost-effective function be associated with each implementation dependent parameter. However, we do discuss which are the dominant aspects of the performance model which we then use for the design of a propose a special-purpose architecture.

Let us consider that the improvement in the average goal reduction time, $\Delta\bar{t}_{red}^i$ due to function i , is proportional to the product of the workload dependent parameter w_i and the change in the implementation dependent parameter $\Delta\bar{t}_i$. In other words, both workload and implementation dependent parameters influence the degree of performance improvement. In the performance model we consider, the implementation of goal suspension and activation is more complex and time consuming, compared to other implementation parameters. Even though a low *system dependent* constant value is associated with the goal suspension parameter, the overall affect on performance can be significant due to

the large value of t_{sup} . The same applies to other goal management operations such as goal activation, creation and termination.

Also shown in the performance model is the number of clause-tries per goal reduction. This parameter significantly affects the clause-try execution time. Using better compilation techniques, the number of clause-tries per reduction is expected to be lower.

5.7 Summary

In this chapter we have defined analytic performance models for the evaluation of the relative execution time of individual functions during goal reduction. The selected functions correspond to previously reported and suspected implementation bottlenecks. We show that the relative execution time of redundant clause-selection during goal reduction may create a serious implementation bottleneck. However, instead of suggesting architectural support for clause selection, we expect this feature to be part of advances in compilation technology. Some preliminary results have been reported in [Klig87].

We conclude that the relative execution time of goal management during goal reduction is very high, and increases with improvements in clause-selection which reduces the average clause-try execution time. In a special-purpose architecture, support for goal management operations is imperative.

The relative execution time of argument dereferencing and variable trailing are less significant than clause-try selection or goal management. However, in a special-purpose architecture where support for both clause-selection and goal management is available, the argument dereferencing and variable trailing execution times may become significant.

We have combined the individual performance models into a general performance model for goal reduction. This allows for changes in a single parameter to be reflected in the overall performance model. Moreover, changing the program parameters, one may consider different workloads in the same performance model.

CHAPTER 6

FCP Processor Architectural Model

We now propose the architectural model of a special-purpose processor for the efficient execution of Flat Concurrent Prolog. In general, a *processor architecture* consists of an instruction set, a storage model and an interpretation mechanism that controls the execution of the processor instructions [Fly88]. By an *architectural model*, we mean the *abstract* functional description of the processor architecture. By *abstract*, we imply that many of the implementation-level details are left unspecified.

The special-purpose FCP processor architecture is proposed with the following considerations in mind:

- *Functional concurrency* is the main form of *intra-processor* concurrency.
- A high-bandwidth memory hierarchy is integrated into the processor architecture to reduce storage access time.

Both of these considerations are represented in Figure 6.1. The FCP processor interpretive mechanism is decoupled and partitioned into multiple functional units that execute concurrently. The functional decomposition of program interpretation into concurrently executing units is motivated by the implementation bottlenecks discussed in Chapter 5. The processor instruction set consists of instructions belonging to each functional unit.

The storage model is defined as a high-bandwidth system by partitioning the storage space into separate memory sections according to the organization of the interpretive mechanism, as well as defining a memory hierarchy that produces a fast response time.

We now describe the FCP processor high-level organization and interpretive mechanism. We then describe in detail the organization and operations of each unit separately.

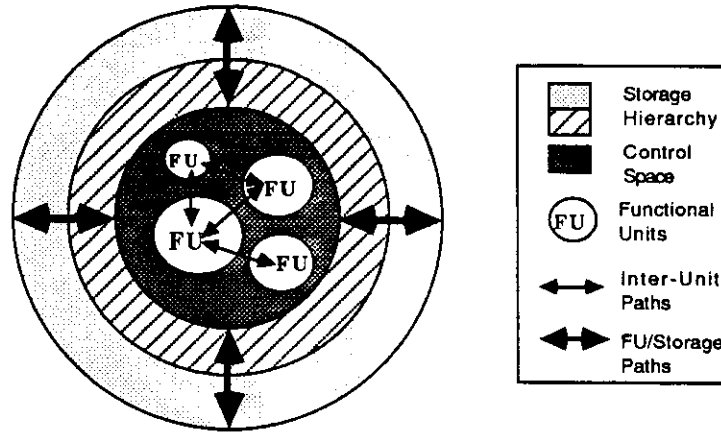


Figure 6.1: Considerations for the FCP Processor Architectural Model

6.1 FCP Processor Top-Level Organization

The FCP processor consists of multiple functional units for the cooperative execution of FCP programs. In Figure 6.2 we show the three-layer hierarchical structure of the processor organization. The first level contains the tightly coupled *execution units*, the second level consists of specialized *cache units* and the third level contains the special-purpose *memory modules*. The following are the FCP Processor concurrent functional units and memory hierarchy components:

1. **Execution Units:** Reduction Unit (RU), Tag Unit (TU), Goal Management Unit (GMU) and Instruction Unit (IU).
2. **Specialized Cache Units:** Goal Cache (GC), Data and Tag Cache (DC), and Instruction Cache (IC).
3. **Memory Sections:** Goal Memory (GM), Data and Tag Memory (DM), and Instruction Memory (IM).

The Reduction Unit, RU, is the main instruction-set unit in the FCP processor. It is tightly coupled with the execution of the Tag Unit, TU. The Goal Management Unit, GMU, serves the Reduction Unit by providing it with a continuous flow of reducible goals and performing concurrent goal management. The GMU executes special-purpose goal management instructions. The purpose of the Instruction Unit, IU, is to service the remaining execution units with special-purpose instructions. RU, TU and GMU are instruction-set units whereas IU prefetches instructions for these units. The operations of the execution units

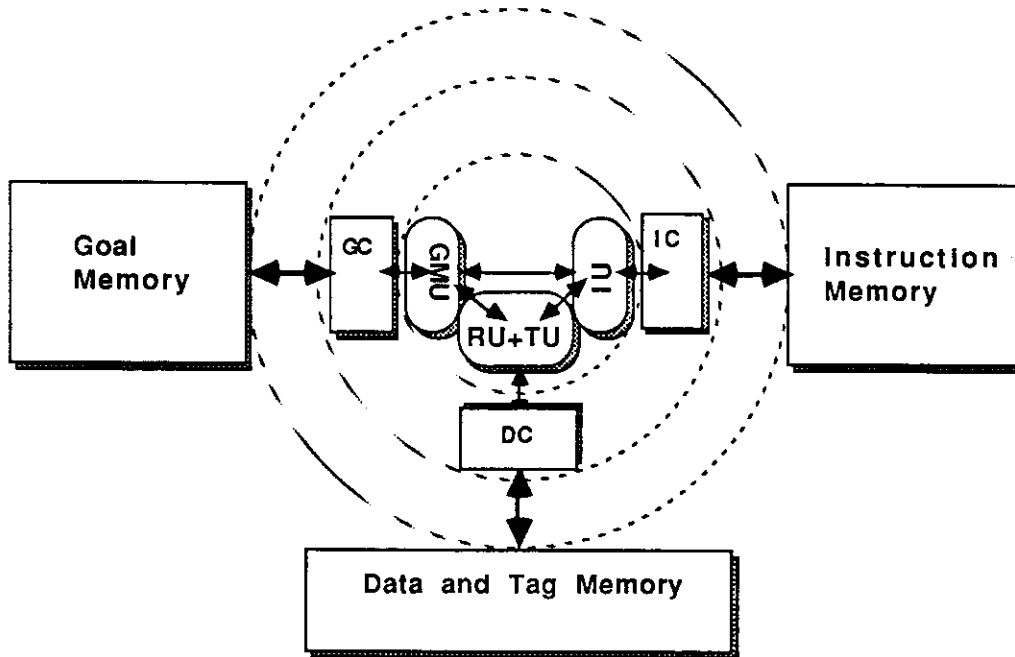


Figure 6.2: FCP Processor Multi-Functional Unit Organization

are dedicated to manipulating specific objects. RU manipulates program data structures, TU manipulates data tags, GMU manipulates goal management data structures and IU manipulates the set of FCP processor instructions.

Three special-purpose cache units are defined as part of the FCP processor architecture. These are the Goal Cache, Data Cache, and the Instruction Cache. The purpose of the dedicated cache units is to enable a higher throughput of objects requested by the execution units. In other words, all execution units access the specialized caches and only if there is a cache miss are objects requested from the dedicated memory modules.

Besides defining multiple specialized functional units and corresponding specialized cache modules, memory is divided into dedicated memory sections. Each memory section is accessed and managed only by the corresponding execution units. The Data Memory is used for storing all program data structures such as lists, variables, tuples, constants ect. The Tag Memory is used to store all the tags of the corresponding data objects. The Instruction Memory is used for storing FCP processor machine instructions and the Goal Memory is used to store all the control structures used for goal creation, suspension, activation and ter-

mination. The Goal Memory is accessed and managed by the Goal Management Unit.

6.2 FCP Processor Instruction Set

The FCP processor instructions are divided into three groups according to the instruction-set execution units: RU, TU and GMU. RU executes a specialized RISC-type instruction set, TU executes instructions that manipulate tag operations and GMU executes goal management related instructions. The corresponding instruction sets are specified in more detail later in this chapter. For now, we are only concerned that each functional unit executes a separate set of dedicated instructions. The dedicated groups of instructions corresponding to each instruction-set execution unit is symbolically shown in Figure 6.3.

FCP Processor Instruction Format

Rather than fetching each execution unit instruction separately and executing them sequentially, we consider one FCP instruction to be of a *wide instruction format*, with dedicated instruction fields corresponding to each functional unit. This instruction format can be seen as a compiler optimization of a sequence of instructions dedicated to separate functional units, as shown in Figure 6.3.

Since the instruction sets for each functional unit vary in number and size, the complexity of instruction fetching and decoding is reduced by defining a uniform instruction format. This instruction format also enables the concurrent execution of instruction fields by corresponding functional units.

Also shown in Figure 6.3 is the Instruction Register (IR) that stores the current instruction. It is fetched by the Instruction Unit from IC, and contains three opcode fields for the RU, TU and GMU functional units. The rest of the instruction is used for storing instruction operands. Since there are very few well defined GMU and TU instructions, and since they do not require additional operands, the operand field is used only by RU.

As a result of compiler compaction of individual functional unit instructions into a wide instruction format, some opcode fields are not filled by useful operations. These are denoted as blank areas in Figure 6.3, and they represent functional unit *no-op* instructions.

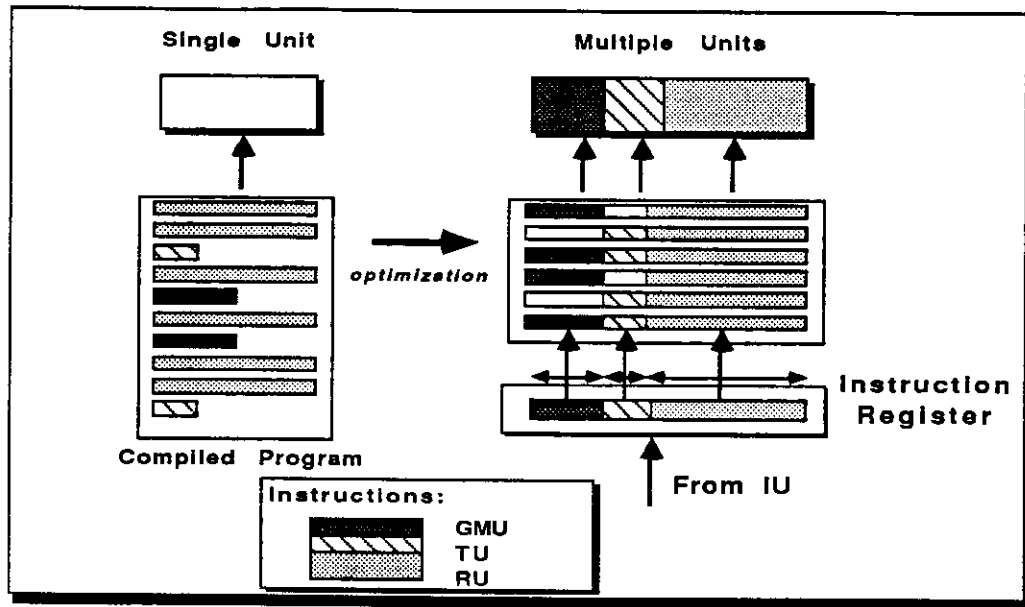


Figure 6.3: Instruction Execution for Multiple Functional Units

6.3 FCP Processor Interpretive Mechanism

FCP program execution consists of successive goal reductions performed concurrently by the specialized functional units. GMU performs all goal-related operations as well as scheduling goals for execution by RU and TU. IU is responsible for providing both RU, TU and GMU with instructions corresponding to the currently scheduled goal.

RU and TU Instruction Execution

RU and TU execute tightly-synchronized instructions that execute in lock-step. For each executed RU instruction, the corresponding TU operation-field may specify a concurrent operation to be performed by the Tag Unit. A new TU instruction is considered together with the following RU instruction. In other words, TU does not execute the following instruction before RU completes its operation. The TU instructions are simple operations with execution times less or equal to the execution time of RU instructions.

RU and GMU Instruction Execution

GMU instructions are high-level operations that take longer to execute than a RU and TU instructions. While GMU executes the current goal management instruction, RU and TU continue executing their subsequent instructions. These instructions are fetched by IU. If the fetched instruction contains another instruction for GMU, prior to GMU terminating the current operation, the processor instruction execution is suspended until GMU completes the instruction.

The overlapped RU-GMU execution is best described using the state diagram shown in Figure 6.4. Each state is defined as a tuple $S_i = \langle s_1, s_2 \rangle$ where $s_1, s_2 \in \{busy, wait\}$ describe respectively RU and GMU in either a *busy* or *wait* mode of execution. We consider the following four states and their transitions:

- $S_1 = \langle idle, idle \rangle$: Both RU and GMU are idle. This is an initial state or a state during program execution in which some external activity, such as user input or system interrupt, is expected. The occurrence of such an interrupt results in a transition to state S_2 .
- $S_2 = \langle busy, idle \rangle$: RU is busy and GMU is idle. RU executes program instructions. If the instruction register IR contains only RU and TU instructions and no GMU instructions, these operations are executed and the state transition remains in state S_2 . If IR contains an instruction for GMU, a state transition is made to state S_3 .
- $S_3 = \langle busy, busy \rangle$: Both RU and GMU are busy. IU continues to fetch instructions executed by RU while GMU executes the goal management operation. If the fetched instruction contains only RU opcode, the instruction is executed by RU. If the instruction contains an opcode for GMU, a transition is made to state S_4 . If during program execution GMU completes its operation before another GMU instruction is decoded, a state transition to S_2 is made.
- $S_4 = \langle idle, busy \rangle$: RU waits for GMU to complete the current operation. Meanwhile the state model remains in S_4 . Upon completion, a transition to state S_3 is made. The pending goal management instruction is then executed by GMU and IU continues to fetch instructions.

Therefore, GMU does not maintain a queue of instructions but can execute only one goal management instruction at a time. An instruction queue for GMU

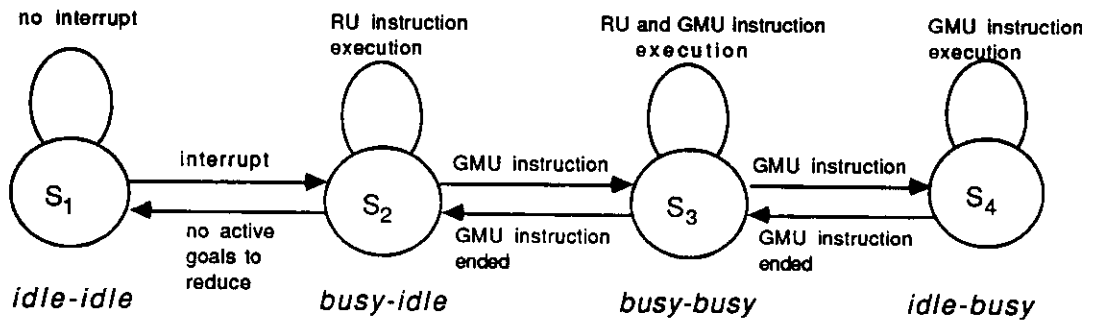


Figure 6.4: State Diagram of RU-GMU Execution Model

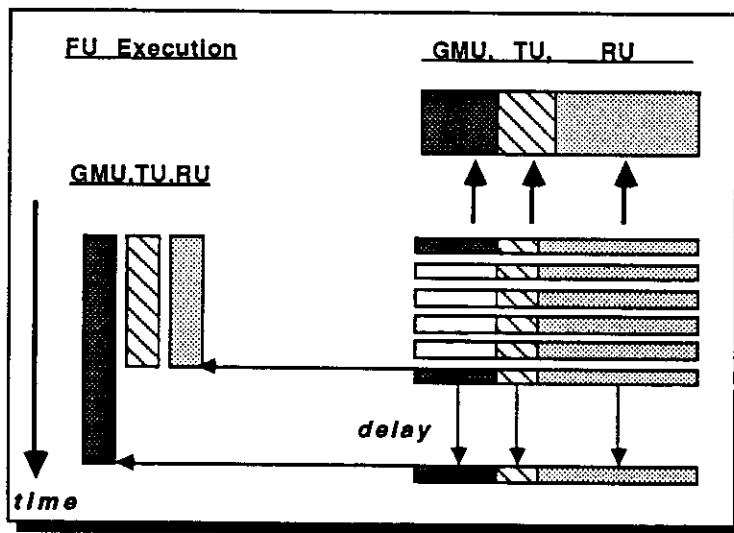


Figure 6.5: RU, TU and GMU Instruction Execution

was initially considered, but the complexity of the corresponding control was too high, since it required the ability to perform program roll-back. Details regarding GMU instruction execution are described in Chapter 7.

In Figure 6.5 we show the overlapped FCP instruction execution, corresponding to the three functional units. Note that the instruction execution is delayed if two consecutive GMU instructions are decoded before the goal management instruction completes its current operation.

We now describe in more detail the characteristics of each execution unit, specialized cache and dedicated memory section.

6.4 Reduction Unit

RU is the main unit in the FCP processor. It is an instruction-set unit with *load/store* RISC instructions and additional specialized instruction support for FCP execution. RU is tightly coupled with other execution units, that is, with GMU, IU and TU. The main function of RU is to reduce *goals* supplied by GMU. A goal is implemented as a data structure that contains a program counter (PC) and argument pointers. RU reduces goals by executing the program instructions denoted by the current goal's program counter. Instructions are requested and received from IU.

The operations of RU are determined by its instruction set. However, in the proposed architectural model for the FCP processor we do not give a fixed instruction set; rather, we define a class of RU instructions. In fact, RU may execute a general purpose instruction set with several modifications that we will discuss in the following subsection. For simplicity, we regard RU instructions as a RISC type instruction set, even though this is not essential.

Before we describe the class of RU instruction sets, we first discuss the set of registers that are accessible by RU during a goal reduction. The actual components that constitute the RU arithmetic and logic units depend on the specifics of its instruction set. We will discuss only those aspects that are essential to support the RU execution model and its interface to other functional units.

6.4.1 RU Register Storage

A complete description of the RU data path requires the specification of all the arithmetic and logic units, as well as their interface to the RU registers. For VLSI RISC processors these issues as well as the tradeoffs involved, are well known and documented. We do not consider this our focus, nor do we offer contributions in this area. Recent commercial RISC processors like the Intel N10 [Inte89], Motorola 88000 [Mele89], MIPS-X [Horo87] or the Crisp [Bere87] as well as university projects [Tayl86] offer a variety of solutions to problems such as: reducing the penalty due to branches in pipelined RISC processors; design of RISC instruction pipeline stages; register allocation; delayed load/store memory slots etc. We assume that RU executes a specialized RISC instruction set at an effective rate of 1 instruction per processor cycle and *borrow*s existing techniques to achieve this throughput.

In Figure 6.6 we show the register address space during a single goal reduction. The following groups of registers are addressable by RU:

- Active Window Registers
- Spawn Window Registers
- General Purpose Registers
- Special Purpose Registers

A register window consists of a fixed number of n registers. For the purpose of describing RU organization and instruction execution, it is not necessary to specify the exact size of the goal window. However, it can be determined according to the average size of a goal; for example, from the measurements presented in Chapter 5, a reasonable number for the goal size is between 7 and 10.

The *active* and *spawn* windows belong to a set of goal windows that are part of a Goal Cache. The role of the Goal Cache will be described later in this chapter. The *active window* contains the active goal currently being reduced. The goal program counter is stored in the first register labeled $a(1)$. Subsequent registers in the active window $a(2)$, ... , $a(n)$ are used for storing the goal argument pointers. The second window is the *spawn window* used for spawning new goals. The spawn window registers are labeled $s(1)$, ... , $s(n)$. We assume that all of the goal windows in the Goal Cache are of equal size.

A small number of general purpose registers are defined, labeled $r(1)$, ... , $r(m)$. As part of the register file design, we assume that the goal window size n is chosen so that it is greater than the average size of a goal, that is, $n \geq (s + 1)$ (average number of arguments plus program counter). Therefore, in most cases one may expect that all of the goal arguments are placed into the goal window registers. Moreover, since the number of arguments, s , is known at compile time, the compiler allocates as general purpose registers all registers that are not occupied by the goal arguments in the active window, that is, registers $a(j)$, $s < j \leq n$. Similarly, the size of spawned goal k is known at compile time so that the registers $s(i)$, $k < i \leq n$ are also used as general purpose registers during a goal reduction. Therefore, in the general case, if there are m general purpose registers, and s is the size of the active goal and k is the size of the goal being spawned by the current goal, then the total number of general-purpose register available to the compiler is:

$$N = (m + 2 \times n - s - k) \tag{6.1}$$

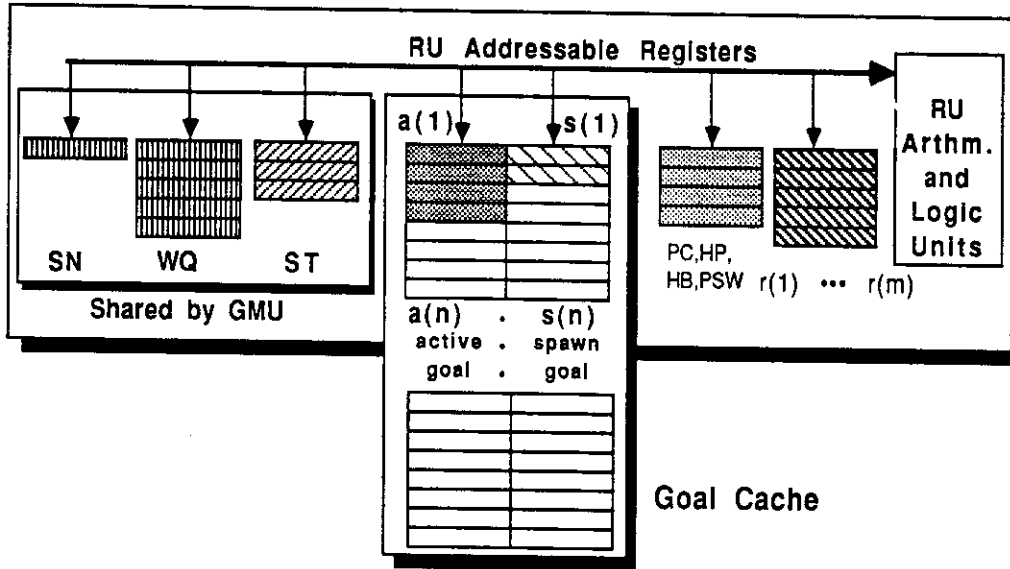


Figure 6.6: RU Addressable Registers During Goal Reduction

for the case where $n \geq (s + 1)$.

However, a program may contain goals that have more arguments than the goal size n . These cases, even though they are expected to be infrequent, are detected at compile time and *rewritten* to have $n - 1$ arguments. This is achieved by compacting the individual arguments that overflow the fixed window size, into a single *tuple* argument. Each element in the tuple corresponds to one goal argument. The compiler then generates the corresponding instructions to extract the arguments from the tuple. This technique is used in the FCP abstract machine compiler.

In addition to the two register windows, and the set of general-purpose registers, RU uses two groups of special-purpose registers. The first group defines the following state registers: Program Counter (PC), Program Status Word (PSW), Heap Pointer (HP) and Heap Backtrack Pointer (HB). The second group of registers are defined to support the goal management operations discussed in more detail in Chapter 7. These include the Suspension Table registers, the WakeUp Queue registers and the Suspension Note register (SN).

6.4.2 RU Instruction Set

The RU instruction set consists mainly of general-purpose instructions with several special-purpose instructions to support FCP program execution. For simplicity, we consider the general-purpose instruction set to be of a RISC type. Only *load* and *store* instructions are used to access memory and all *register_to_register* instructions execute in a single processor cycle. There is another reason to consider RU instructions as RISC. Since the FCP processor consists of multiple functional units that each perform specific (some of them high-level) operations during program execution, RU is left to perform simple operations. Moreover, defining RU instructions as RISC, aides the overall feasibility of designing a single-chip multi-functional unit VLSI processor.

We now discuss the following groups of instructions that contain special-purpose support for FCP program execution.

- Special Load and Store Instructions
- Branching Instructions
- Goal Management Support Instructions

Load and Store Instructions

To support pointer dereferencing, a special mode of the *load* instruction called *load(SR,DR)* is defined that continuously follows the chain of references until a non-reference value is found. Starting with the address stored in register SR, the first word of the dereferenced value is stored in register DR. In addition the address of the dereferenced value is stored in SR. By previously moving the value in SR to a register R and by following the dereference operation *load(SR,DR)* with the *Store(R,SR)* instruction, the effect of *reference shortening* is achieved. All following references to the address stored in R will immediately reference the final value rather than following the previously traversed reference chain. This is particularly useful in languages like FCP where lengthy chains of references can easily be created by repeated variable-to-variable unifications. For example, in the FCP benchmarks considered in Chapter 5, dereference lengths of over 400 were detected.

The execution time of the *load(SR,DR)* instruction is linearly proportional to the dereference length. In Chapter 5, we showed that the average dereference

length in the selected FCP programs is 2.1. This instruction is also aided by the TU that decodes the data tag upon its loading from memory. Therefore, there are no explicit instructions to check the tag and branch if it is a reference.

A special-purpose dereference instruction was also used in the *Carmel* RISC processor for FCP [Hars88]. Similar memory reference behavior was also reported for Prolog in [Tick85].

Branching Instructions

In FCP, operations are commonly determined based on the operand type. Thus, dereferencing a pointer is followed by a test-and-branch operation on the type of the dereferenced value. This is analogous to a *case* statement. Rather than having multiple compare and branch instructions, we define a multiple switch(b1,b2,b3,b4) instruction that branches on the four most commonly used data types.

In [Hars88] it is claimed that all data types in FCP are of equal probability. Our measurements disagree with this observation, at least for the workload that we consider. In Table 6.1 we show the *spectrum* of dereferenced data types found in the selected FCP benchmark programs. The most frequently dereferenced data type is the logical variable (Var), followed by tuple (Tup), string (Str) and integer (Int). In all of the selected FCP programs, these four data types accounted for 75% of all the dereferenced objects. Therefore, the four branch addresses in the switch instruction are implicitly defined for the four types: Variable, Tuple, String and Integer.

Once again, the results represent a specific workload and are not necessarily applicable to other workloads. If an additional test is required, for the types that are not implicitly included in the switch instruction, (for example the *nil* data type), a separate instruction sequence is used.

Goal Management Support

The following two instructions support the execution of GMU overlapped goal management operations. These are the *load* and *move* instructions that have as a destination the Suspension Table (ST) and the Wakeup Queue (WQ). Since ST and WQ are managed as stacks, each has associated a stack pointer value.

Type	FCP Benchmark Programs						
	Compiler	Sim1.	Sim2.	Debug.	Solver	Distr.	Logix
Var	0.23	0.32	0.21	0.37	0.23	0.38	0.30
RO Var	0.05	0.12	0.05	0.03	0.17	0.07	0.08
Integer	0.17	0.05	0.04	0.08	0.08	0.05	0.10
Real	0.00	0.00	0.00	0.00	0.00	0.00	0.00
String	0.15	0.16	0.23	0.13	0.12	0.14	0.16
Nil	0.03	0.01	0.02	0.01	0.03	0.01	0.02
Car Ref	0.13	0.06	0.10	0.04	0.08	0.05	0.07
Car Int	0.07	0.01	0.02	0.01	0.07	0.00	0.05
Car Nil	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Tuple	0.16	0.26	0.32	0.35	0.22	0.30	0.21
Vector	0.01	0.01	0.02	0.00	0.00	0.02	0.01

Table 6.1: Distribution of Dereferenced Data Types

Successive writes into the corresponding stack increments the stack pointer.

Since ST and WQ consist of a finite number of registers, exception conditions are detected when either of the two overflow. In this case, the goal that caused the condition is restarted, but now using the Data Memory instead of ST and WQ. For example, if more goal pointers need to be stored in WQ than there are available registers in WQ, the activation of goals is performed incrementally, under the control of the exception handler.

6.5 Tag Unit

FCP incorporates polymorphic operations on primitive data types. As a result, data values are distinguished using tags. All data tags are stored in a separate memory section called Tag Memory, (TM). Architectural support for tag related operations is not a new feature and is common in implementations of symbolic languages such as Lisp [Moon85], [Tayl86]. Moreover, all of the Prolog processors as well as the *Carmel* processor for FCP, described in Chapter 3, include special-purpose support for tag processing as part of the machine instruction set.

In the FCP processor, all tag operations are performed by TU. A separate TU-TM processor to memory path for tags enables concurrent tag access. We

now describe the types of operations performed by TU.

TU Instructions

As described in [Stee87], run-time tag operations can be grouped into the following four types of instructions:

- Tag Insertion
- Tag Removal
- Tag Extraction
- Tag Checking

Tag *insertion* consists of setting the data type of an operand. Given the operand *value*, the $\langle \text{tag}, \text{value} \rangle$ pair is formed by setting the corresponding *tag* part. The reverse operation is called *tag removal*. Given the tagged object, only the value is required while the tag is removed. Tag *extraction* is performed when the tag value is needed for further processing. For example, in the tag *checking* type of operation, the tag value is compared to another tag.

TU performs the four types of tag operations concurrently with RU execution in a tightly coupled fashion. Tag setting is performed concurrently with the $\text{store}(\text{SR}, \text{DR})$ instruction executed by RU. While the value in SR is being stored in the data memory at location denoted by DR, the tag type denoted by the $\text{set}(\text{Tag})$ instruction is stored in the same memory address in the Tag Memory. The effect is as if the $\text{store}(\text{SR}, \text{Tag}, \text{DR})$ is executed as a single instruction. For example, to store the contents of SR into the address denoted by DR and to set its tag value to integer, the instruction mnemonic would be: $\text{store}(\text{SR}, \text{Int}, \text{DR})$.

Similarly, on each $\text{load}(\text{SR}, \text{DR})$, the tag value corresponding to the source address is fetched together with the data value, and stored in the Tag register. This corresponds to both tag *extraction* and *removal*. In addition, the fetched tag value is decoded and sets the *type condition codes* that are accessible by RU. This corresponds to tag *checking*. Using branch type of instructions, RU can check the data types. In Figure 6.7 we show both the tag setting, and tag loading and decoding operations.

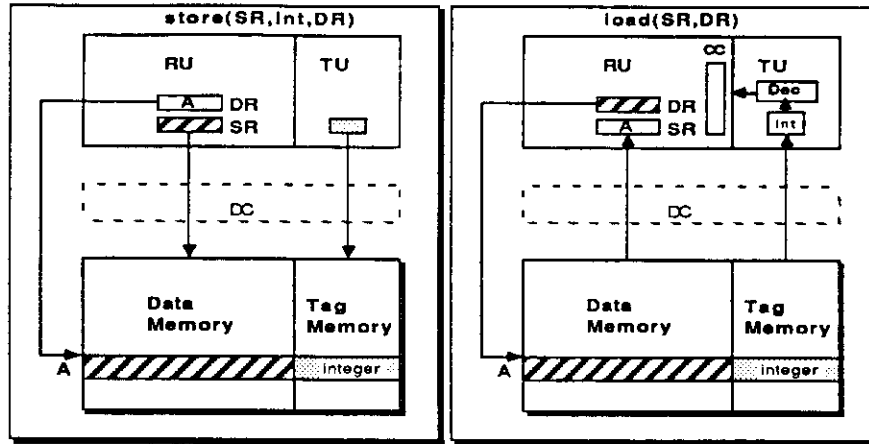


Figure 6.7: TU Tag Setting and Loading

6.6 Goal Management Unit

In FCP and in other concurrent logic programming languages, spawning, halting, suspending and activating concurrent goals represents the main control mechanism analogous to procedure calls or *co-routines* in procedural languages (see Summary of Chapter 3). The purpose of GMU is to reduce the effective time spent performing goal management operations. By effective time, we imply the time as seen by RU, which includes the overhead of communication and synchronization. If we consider a processor organization that does not have any architectural support for goal management, then, the effective execution time corresponds to the time it takes to execute processor instructions that perform the specified functions. We propose to reduce this effective time in the following way:

- GMU operations execute concurrently with goal reduction performed by RU and TU. In other words, an overlapped goal reduction and goal management execution model is defined.
- Goal management operations are complex, high-level operations that may take many RU instruction cycles to execute. To implement goal management operations efficiently, a special-purpose Goal Cache is used.

Both of these issues are the focus of Chapter 7. We now discuss the GMU organization and instruction execution.

GMU Organization

In Figure 6.8 we show the organization of GMU. It consists of the following components:

- Goal Management Controller (GMC).
- GMU Registers (GMR).
- Goal Status Bits (GSB).
- Memory Port (GMP) to Goal Memory

GMC controls the execution of GMU instructions received from IU. In addition, it shares one control flag called *Busy Flag* used by IU to synchronize RU and GMU execution. When GMU receives an instruction from IU (assuming GMU in not busy), the GMU *Busy Flag* is set. Subsequent FCP processor instructions that contain GMU operations are suspended until the GMU *Busy Flag* is reset. When GMU completes its current operation, it resets the flag. Any pending instruction is then resumed.

GMU contains five special-purpose registers used for managing the Goal Memory (GM). These are the Heap Pointer (HP), Goal Queue Front (QF), Goal Queue Back (QB), Goal Free List (GFL) and Suspension Free List (SFL). The use of these registers is discussed later in this chapter and in Chapter 7.

The goal status bits, GSB, reflect the status of the goals stored in the Goal Cache, GC. These are used by GC to implement the goal management operations. The separate memory port, GMP, to GM allows the concurrent transfer of goal data structures between GC and GM.

In addition to the above components, GMU shares hardware resources with RU. The control for these resources is managed so that their access is mutually exclusive. GMU requires the access to the following shared data:

- Goal Window Pointers: Current Goal (CWP), Current Spawn (CSP), Next Goal (NGP) and Next Spawn (NSP) window pointers.

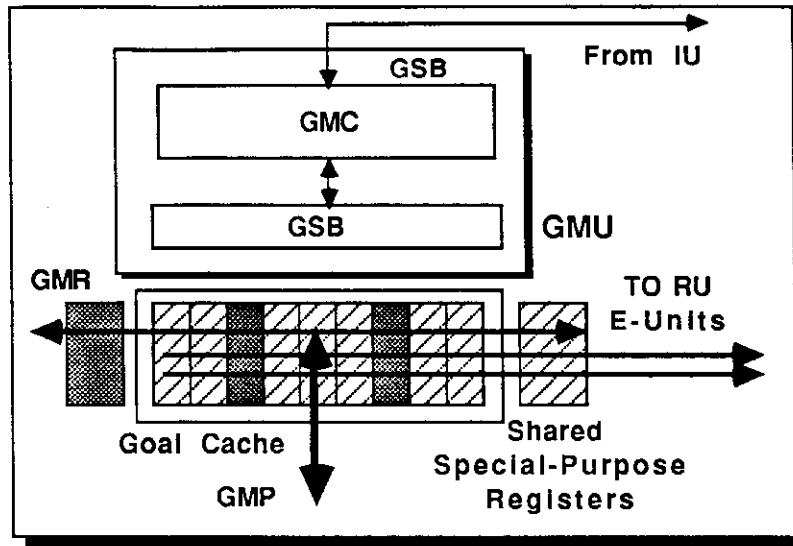


Figure 6.8: GMU Organization

- Variable Suspension Table (ST).
- Wake-up Queue (WQ).
- Suspension Note (SN).

The goal window pointers are used by GMU to manage goals in GC. The pointer values are modified in a mutually exclusive way by both RU and GMU. The variable Suspension Table (ST) and the Suspension Note register (SN) are used to implement the goal suspension algorithm. This is fully described in the following chapter. The Wake-Up Queue (WQ) is used to store pointers to goals that should be activated if the clause-try commits.

GMU Instruction Execution

In the FCP processor, GMU executes the following special-purpose goal management instructions: halt, spawn, suspend and commit. All goal management instructions are implemented using the Goal Cache. Those goal management instructions that result in a *cache hit* are implemented by simply changing the value of the goal status bits. However, all goal management instructions that

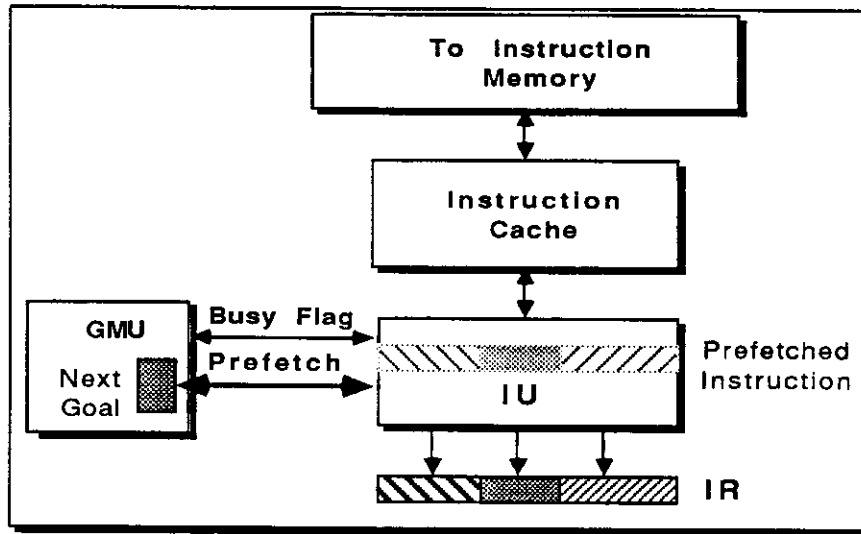


Figure 6.9: IU Instruction Prefetching

require access to the goal memory are interpreted as a goal *cache miss*. The organization of the Goal Cache is described in section 6.8.

6.7 Instruction Unit

IU fetches instructions stored in IM and cached in IC. The fetched instruction contains operation fields for each of the execution units and is stored in the Instruction Register, IR. The sequencing of instructions depends on the value of the RU program counter, PC, but also on the status of GMU. In Figure 6.9 we show that IU shares the *Busy Flag* with GMU. If this flag is set, and if the GMU opcode is not empty, IU suspends the RU pipelined execution until the flag is reset.

Besides fetching instructions, IU also performs the following prefetching operation. As it will be described in the following section (and in more detail in Chapter 7), the Goal Cache always has at least one goal prefetched into a free window, in order to enable fast goal switching. The first register in the next-goal window is the program counter of the next goal. This value is used by IU to prefetch the next instruction prior to the goal switch. Upon a goal switch, the next instruction is immediately placed into the instruction pipeline, allowing for

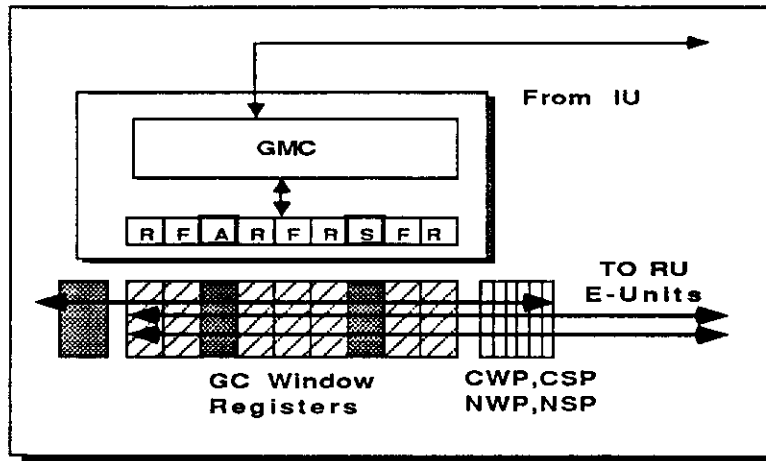


Figure 6.10: Goal Cache Organization

zero-cycle switching between goals.

6.8 Goal Cache

GMU performs efficient goal management by manipulating goals in the goal cache. The goal cache consists of N goal windows implemented using a fixed number of processor registers, as shown in Figure 6.10. At each point during program execution, there are two windows addressable by RU: the *Active* or the Current Goal Window (CGW) and the Current Spawn Window (CSW). These windows are denoted by the pointers CWP and CSP, which are part of the RU Program Status Word register (PSW). CGW contains the currently executing goal while CSW is empty and is used for spawning a new goal. Each goal-window in GC may be in one of the following four states:

- *Active*: window contains the currently executing goal.
- *Spawn*: window is empty and is currently being used for spawning.
- *Ready*: window contains a goal that is ready to be scheduled for execution.
- *Free*: window is empty.

If GC contains more than a single *ready* goal, switching between goals is performed within GC and involves changing the value in the CWP and updating the goal status bits. Placing a newly spawned goal onto the active goal queue consists of changing its status to *ready*. Similarly, halting an active goal is implemented by changing its status from *active* to *free*. All goal-window states are stored in the Goal Status Bits (GSB) module. Moreover, all operations on the status bits are also performed in the GSB.

The Goal Management Controller (GMC) detects two exceptional conditions in GC. First, GC *overflow* occurs when the goal cache becomes full and there is no place available in GC for spawning. Upon overflow, a goal is moved to the goal queue in GM. The front and back of the goal queue are denoted with the GQF and GQB goal memory pointers. To implement efficient spawning of new goals, GMC detects overflow before the GC becomes completely full. That is, GMC ensures that there is always one empty goal window available in GC for fast spawning.

The second exceptional condition is GC *underflow*, which occurs after a sequence of halt instructions deplete GC of available goals for scheduling. To implement efficient halting, GMC ensures that there is always at least one prefetched goal available in GC. This enables GMU to always schedule a new goal whenever the current goal is halted. Otherwise, RU may need to wait until a new goal is fetched from GM.

Goal Cache Policy

A detailed description of the GC operations is presented in Chapter 7; here we describe the main concepts. Goal scheduling is determined by inspection of the goal window status bits stored in GSB. In GC, the first *ready* goal is scheduled for execution and the goals are selected in a round-robin fashion relative to the halting goal. Spawning a new goal is performed in one of the available windows that is labeled as *free*. This is analogous to a *cache placement policy*. If there are several windows that are empty and marked as *free*, the first empty one is selected.

In case of a goal cache overflow, one goal is moved from the goal cache into the goal memory. The selection of the goal is analogous to the goal cache replacement policy. The first non-empty goal window is selected. In the implementations that we consider, we are more concerned with implementation efficiency than with a

certain scheduling strategy.

Goals are moved from the goal memory into the goal cache only when goal cache underflow occurs. When a goal suspends, it is removed from GC unless it is one of only two goals in GC and the goal queue in memory is empty. This avoids the unnecessary thrashing of the same goal.

An important concern when considering goal scheduling is *fairness*. By using a goal cache, we do not implement *strict fairness* of goal scheduling, since a recently spawned goal is scheduled before goals that were spawned before it. Actually, the GC policy is designed for efficiency rather than fairness. On the other hand, to prevent goals in GC from being scheduled indefinitely, we use a *time-out* mechanism. After a certain number of goal reductions in GC without any goals being moved to the GM (overflow) or brought in from the GM (underflow), a GC *miss* is induced if the goal queue in GM is non-empty.

It is important to note that we emphasize the flexibility of the design approach of the GMU and do not claim an optimal goal scheduling policy. The prospect of *priorities* amongst goals can also be considered and is discussed later in this thesis.

6.9 Data Cache

The Data Cache (DC) is used for storing both data and tag values. In this section we consider a special data cache policy that supports clause-try trailing. We refer to this cache policy as *Delayed Binding*. It can be described as follows:

- *All assignments performed during a clause-try are 'delayed' until the outcome of the clause-try is known. If the outcome is successful, the bindings become permanent otherwise they are cleared.*

By *delayed* bindings, we imply that the bindings are considered as temporary while the outcome of the clause-try is not known. If the clause-try commits, the bindings become permanent, otherwise they are ignored.

The Data Cache policy provides architectural support for shallow backtracking. Two other techniques have been proposed to support shallow backtracking in Prolog. First, in [Tick87], the use of a choice-point buffer is described that keeps the top of the choice-point stack. In the Pegasus VLSI RISC processor

[Seo87], shadow-registers are used. When a choice point is encountered, the current choice point is moved to the shadow registers and saved during idle memory cycles.

The approach in [Tick87] suggests the use of special-purpose buffers, that could be used in addition to a Data Cache. On the other hand, the Pegasus approach requires significant processor resources to be allocated as part of the processor register file. The method proposed for the FCP processor assumes the existence of a Data Cache and considers modifications to the cache policy to support shallow backtracking. We now discuss the Data Cache policy in more detail.

Data Cache Assumptions

The following assumptions are made regarding the Data Cache and its interface to RU and TU:

- The Data Cache distinguishes *read*, *write* and *trail* requests from RU.
- Two control signals from RU affect the status of the cached elements. These are the *commit* and *fail* control signals.
- Changes in the cached data status are performed atomically and efficiently by modifying the status bits.

Given the above conditions for the Data Cache, we now define the following cache policy for shallow backtracking.

Data Cache Algorithm With Trailing

Let us assume that the data cache is fully associative with a write-back policy. The data cache stores the address, the value, tag and status of the cached elements. The status is defined to be: *Empty*, *Clean*, *Dirty* or *Trailed*. An entry labeled *Empty* is vacant. If the status is *Clean* the cached element is identical to the corresponding value in DM. A *Dirty* status indicates that the stored value in the cache differs from the value in memory, and the *Trailed* status indicates that the value in the cache is temporary.

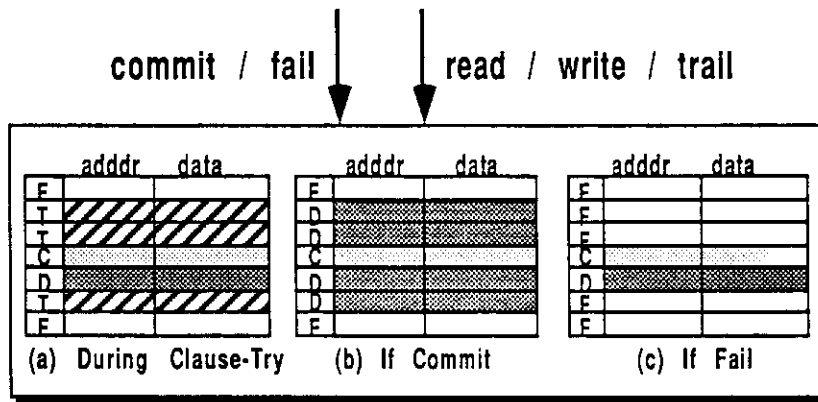


Figure 6.11: Data-Trail Cache Policy: Delayed Binding

DC receives *read*, *write* and *trail* memory requests. The read and the write requests are treated in the conventional way. Upon receiving the trail memory request, DC performs the following operation:

- **trail(Address,Value)**: If there is a cache hit, the following cases may occur depending on the status of the cached element. If it is *Clean*, the new value is stored in the cache and marked as *Trailed*. If it is *Dirty* the cached value is written back to the DM and the new value is stored in the cache and marked as *Trailed*. If it was already *Trailed*, the new value is written in the cache and remains *Trailed*.

In case of a cache miss, the cache replacement policy vacates an entry that is not *Trailed*, writes the new value in the cache and marks it as *Trailed*. If the non-trailed element is not found, an exception condition is generated.

The following operations are performed when the control signals *fail* and *commit* determine clause-try failure or success.

- **fail**: All *Trailed* entries are marked Free.
- **commit**: All *Trailed* entries are marked Dirty.

In Figure 6.11a we show the DTU cache during a clause-try. The elements marked as *T* are being trailed. In Figures 6.11b and 6.11c we show the contents of the cache after a clause success and failure. The complete data cache algorithm is specified in Figure 6.12.

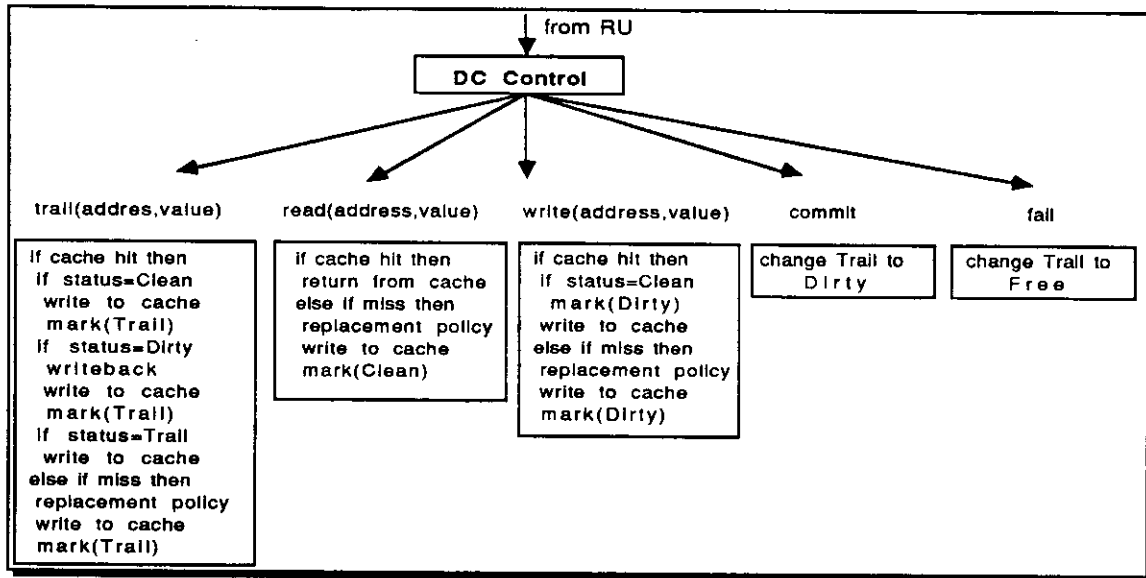


Figure 6.12: Data Trail Policy

Therefore, the proposed data cache policy implements the Delayed Binding approach to data trailing by keeping the trailed values in the cache and either committing them to Dirty upon clause-try success, or resetting them upon clause-try failure. One should note that Trailed values are never replaced by the cache replacement policy. Furthermore, trailed values are accessible during the clause-try even before they commit or fail. Trailed values that commit to Dirty remain in the data cache as valid cache entries.

The following features of the Data-Trail Unit and the Data Cache policy should be discussed. First, there are no explicit instructions used for saving, that is trailing assignments in the Data Memory. This is implicitly performed as part of the Data Cache operation. Second, restoring a previous memory state due to clause-try failure can be performed in constant time setting the appropriate bit in the status of the cached elements. And finally, at commit time, the trailed elements remain in the cache thus improving the locality of data references.

The Data Cache policy for shallow backtracking, relies on the cache replacement policy not to replace any trailed values from the cache. This is because the trailed values are temporary, and should thus not reach the Data Memory, where they will overwrite the existing contents of memory. The exception condition that detects the case where all of the elements in the cache are trailed, results in

the use of a Trail Stack in the Data Memory. This is performed by the exception handler. If the Data Cache is fully associative, such conditions are expected to occur very infrequently. From the measurements shown in Chapter 5, clause-tries that commit, in most cases require trailing of less than 10 elements.

Since a fully associative cache is costly, a set associative cache may be more suitable for implementation in the FCP processor. In this case, the same *Delayed Binding* approach for shallow backtracking applies to the set associative organization, as well as other cache organizations, as long as the trailed values are not replaced and written back into memory. The chances for the exceptional condition to occur is higher in cache organizations that are not fully associative.

6.10 Memory Modules

The address space of an FCP program is partitioned into four areas: *Instruction*, *Goal*, *Data* and *Tag Memory*. The compiled program is stored in IM which is accessed and managed by the IU. GM is used for storing all of the goal control structures. It is accessed and managed only by GMU. Tags are stored in TM whereas DM is used for storing program data structures like lists, variables, tuples, integers etc.

The data and tag memories are accessed using the common address. Memory is allocated on the heap using the HP and HB pointers. The stop and copy algorithm for garbage collection could be used to recycle discarded data structures.

6.10.1 Goal Memory

GM is accessed and managed solely by GMU. It is used for storing goal control structures private to GMU. GM contains three types of control structures as described in Chapter 4. These are: *goal records*, *activation records* and *suspension records*. Memory allocation in GM is performed either by using the Goal Heap Pointer (GHP) or the two free list pointers: goal free list (GFL) or suspension free list (SFL). GFL are used to link discarded goal structures in GM. Similarly, activation and suspension records are linked onto SFL.

When a goal suspends, associated with each goal record is a unique *goal hanger* that consists of the goal pointer and a reference count of the number of variables the goal is suspended on. This allows the activation record to be

dynamically garbage collected once there are no more variables that point to it (this avoids having dangling pointers). Therefore, GM is dynamically managed independently of Data Memory or any other memory section.

6.11 Chapter Summary

The FCP processor architectural model defines multiple functional units that execute concurrently. Based on the performance analysis models described in Chapter 5, we have proposed architectural support for each of the hypothesized implementation bottlenecks.

The FCP processor instruction set enables each functional unit to receive instructions concurrently, thus resulting in a *long instruction format*. The Goal Management Unit offers architectural support for overlapped goal management and efficient execution using a special-purpose Goal Cache. Zero-cycle switching between goals is enabled by prefetching instructions from the next goal already in the Goal Cache. Instruction prefetching is performed by IU. The Reduction Unit and TU perform goal reduction by executing a specialized RISC instruction set. A separate instruction provides support for pointer dereferencing. Clause-trailing during shallow backtracking is supported by the specialized Data Cache. For this purpose, the cache policy called *Delayed Binding* is defined.

CHAPTER 7

Overlapped Goal Reduction and Goal Management

We now consider the overlapped execution of goal reduction, performed by RU, with goal management performed in GMU. We divide this chapter into the following five sections. First we describe how to decouple the two sets of operations, starting from the sequential abstract machine for FCP, described in Chapter 4. In the second section we describe how RU interprets goal management operations that execute in GMU. In other words, we describe the RU-GMU synchronization and interface. In the third section we present the goal management execution algorithms and describe their implementation using the special-purpose Goal Cache. In the fourth section we give several examples of GMU execution, including goal cache overflow and underflow. Finally, in the fifth section we summarize the properties of the overlapped execution algorithms.

7.1 Overlapped GMU Execution

Starting from the sequential abstract machine for FCP, and for each of the goal management instructions: halt, spawn, suspend and commit, we first describe the dependency between these operations and goal reduction execution. We then suggest a way to remove the dependencies, decouple goal reduction from goal management, and allow overlapped execution in a special-purpose environment.

Halt

In the sequential abstract machine described in Chapter 4, goal termination or halting consists of scheduling a new goal for reduction and also performing some memory management operations. There is no data dependency between the goal management instruction halt and the goal reduction of the next scheduled goal, which corresponds to the get and unify instructions. That is, if the first instruction of the next goal is prefetched, the two operations can execute concurrently in a special-purpose environment. In Figure 7.1(a) we show this by

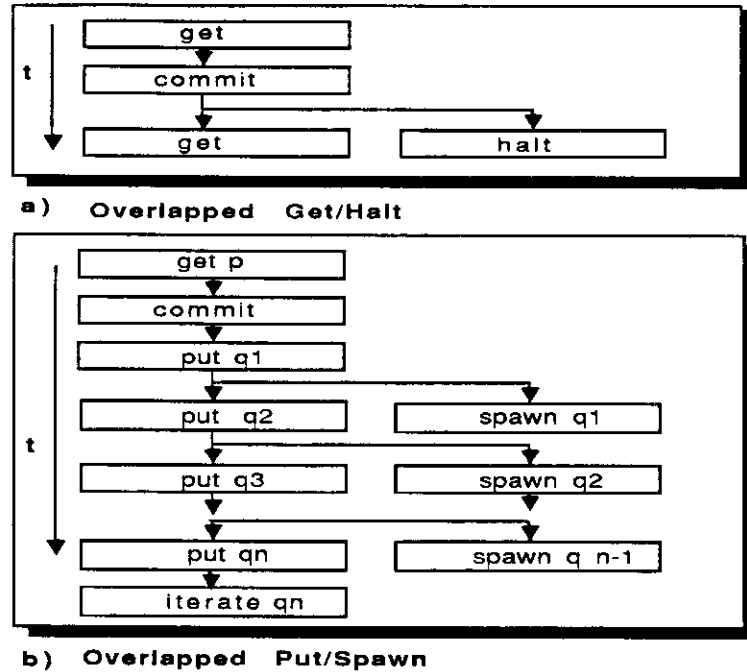


Figure 7.1: Overlapped: a) get/halt; b) put/spawn

representing the two operations halt and get as overlapping.

Spawn

Spawning a new goal consists of placing the goal record onto the active goal queue from which it will be scheduled for execution. In the sequential abstract machine, the spawn operation also allocates the new goal record. During the execution of a clause-body with more than one goal, there is no data dependency between the actual scheduling of the goal record and the creation of the goal arguments of the following goal in the same clause. Therefore, the put and spawn operations of successive goals in the clause-body could overlap, as shown in Figure 7.1(b).

Suspend

During a clause-try, the get and unify instructions use the Suspension Table (ST) to store the address of variables that the goal may suspend on. If the outcome of a clause-try is indeed *suspend*, ST is used to implement goal suspension.

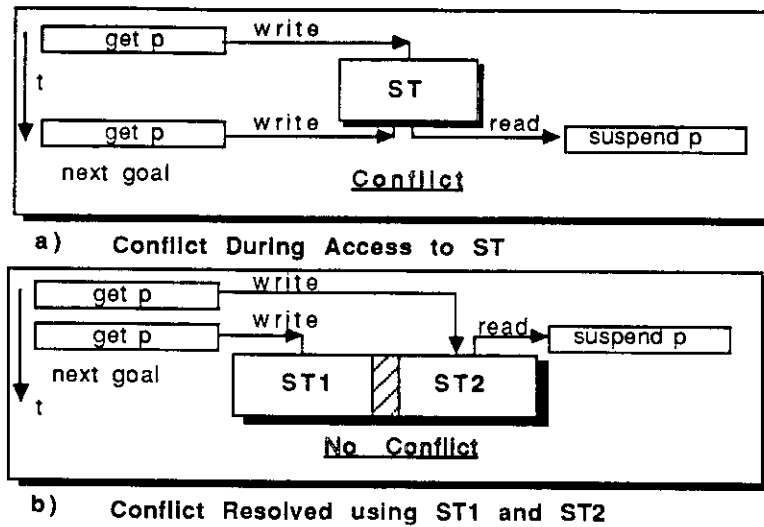


Figure 7.2: Enabling the Overlapped Execution of Suspend

The next goal in the active goal queue is then scheduled for execution. The suspend instruction of the current goal cannot overlap execution with the get and unify instructions of the next goal reduction, since they may both access the same ST structure. Therefore, there is an implicit data dependency because of the use of the same shared resource, ST. We show this conflict of access in Figure 7.2(a).

One way to allow the overlapped execution of the suspend operation and the clause-try instructions of the next goal is to partition ST into two parts. One is used by RU and the other by GMU. The data dependency is avoided by moving the contents of the ST that is accessed by RU into the one accessed by GMU. The effect is the same as if one were to consider the use of two suspension tables ST1 and ST2 used by the get and suspend instructions in an alternating manner. Upon goal suspension, goal reduction continues with the alternate ST, while goal suspension is implemented concurrently using the old ST. Whether a *switch* or a *transfer* mechanism is used is an implementation issue not discussed here. This is shown in Figure 7.2(b).

Commit

The commit instruction consists of activating previously suspended goals that have received new assignments during the committed clause-try. In Figure 7.3(a)

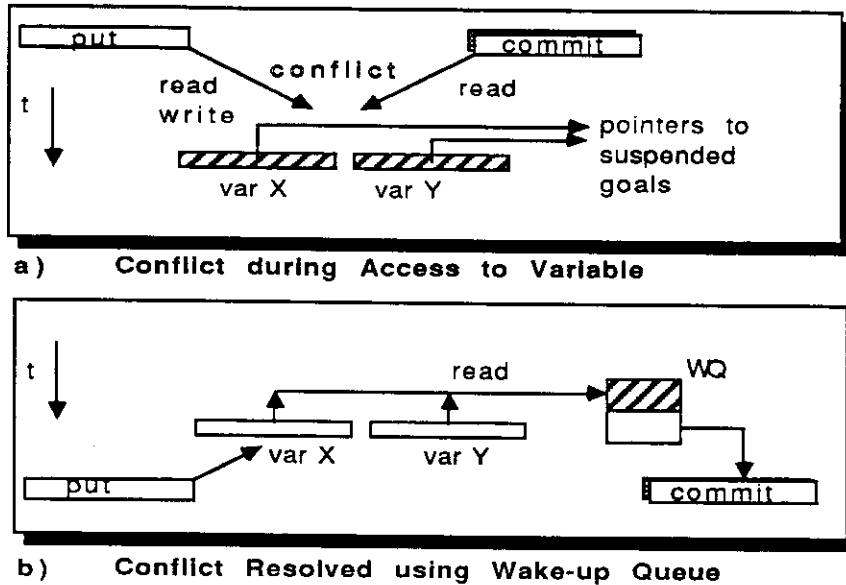


Figure 7.3: Enabling the Overlapped Execution of Commit

we show two shared variables X and Y that have goals suspended on them. This is implemented by storing a goal pointer in the variable location. The commit instruction encountered during the reduction of a non-empty clause-body is commonly followed by put and unify operations. Goal reduction may not overlap with goal activation since the unify instructions may overwrite the same variables that contain suspension record pointers required to activate suspended goals. Therefore, there is a conflict between the commit instruction and the put and unify operation performed in the clause-body of the committed clause.

In Figure 7.3(b) we show the use of a Wake-up Queue (WQ) to store the suspension record pointers to suspended goals. Prior to the commit instruction, the goal pointers are stored in WQ. In a way similar to the use of ST, WQ is partitioned into two parts accessed separately by RU and GMU. The commit operation consists of switching to the new WQ section that is empty while GMU accesses the part containing suspension record pointers. Therefore, overlapped goal reduction and goal management is enabled by the use of two STs and WQs for overlapped goal suspension and goal activation respectively. On the other hand, there is no data dependency between goal termination, goal spawning, and goal reduction. In Figure 7.4 we symbolically denote the overlapped execution of goal management and goal reduction.

Prior to suspension, ST contains variable entries that GMU uses to imple-

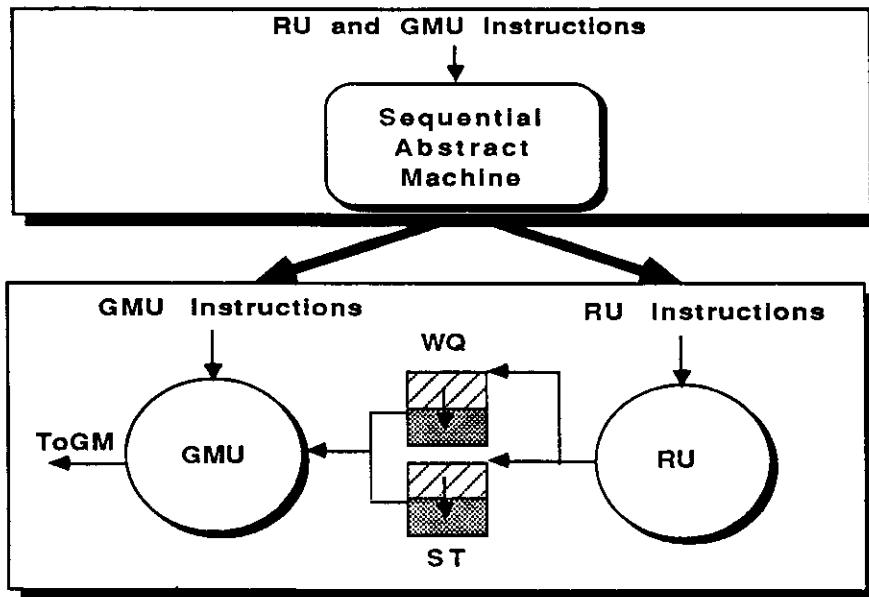


Figure 7.4: Overlapped Goal Management and Goal Reduction

ment the goal suspension mechanism. These were moved into ST by executing specialized RU instructions. At suspension, the context of ST is transferred so as to vacate the current ST, and allow GMU to access the old ST. While GMU performs goal suspension, RU continues to execute instructions. During goal suspension, the WQ is unaffected.

Prior to commit, RU stores in WQ suspension note pointers that GMU uses to activate suspended goals. The pointers are stored in WQ using the same *push_WQ* instructions in the same way as described for ST.

7.2 RU-GMU Interface

The FCP processor instruction stream can be described as a sequence of specialized *RISC-like* instructions executed by RU and interleaved with high-level goal management operations. In a general-purpose execution environment, the goal management operations are also encoded using the host machine instructions. Moreover, the goal reduction instructions and the goal management instructions would execute sequentially rather than in an overlapped mode. The RU instructions that follow goal management instructions may either belong to

the same goal or may result in a switch to another goal. In case of a switch, RU instructions are fetched from the next scheduled goal. This is analogous to a *context switch* in a multiprocessing system.

In the execution environment that we propose, goal management operations execute in an overlapped mode with goal reduction. Furthermore a special-purpose goal cache enables their efficient implementation. In the previous section we showed how the goal management operations are decoupled from goal reduction operations using a special-purpose suspension table and wake-up queue. From the RU point of view, the effective execution time of goal management operations is measured in terms of elapsed instruction cycles between the RU instruction preceding and following the goal management operation. For example, in a sequential environment, the goal management instruction can be implemented as a subroutine call. Only after the call is executed and returns control, is the next RU instruction that follows the goal management operation, executed. In this case, the effective execution time of the goal management instruction is equal to the number of cycles required to make the call, perform the operation, and return control.

In the following section we describe the mechanism used to achieve a zero-cycle delay, or zero cycle effective execution time, for the interpretation of goal management instructions by RU.

7.2.1 Zero Cycle Delay

RU addresses the current goal window (CGW) and the current spawn window (CSW) in the goal cache using the current window pointer (CWP) and current spawn pointer (CSP) respectively. In addition, GMU distinguishes two windows and window pointers in the goal cache: the Next Goal Window (NGW) and the Next Spawn Window (NSW) are denoted using the next window pointer (NWP) and the next spawn pointer (NSP). NWP points to the next *ready* goal in the goal cache whereas NSP points to the next *free* window that will be used for spawning. The next pointers are set by GMU, but read by RU. In other words, the RU-GMU interface that allows the efficient interpretation of goal management instructions consists of managing four goal window pointers. This is shown in Figure 7.5. The current window pointers CWP and CSP are modified only by RU.

From RU's point of view, goal management operations are performed by moving the next pointer values into the current values. In Figure 7.6 we show

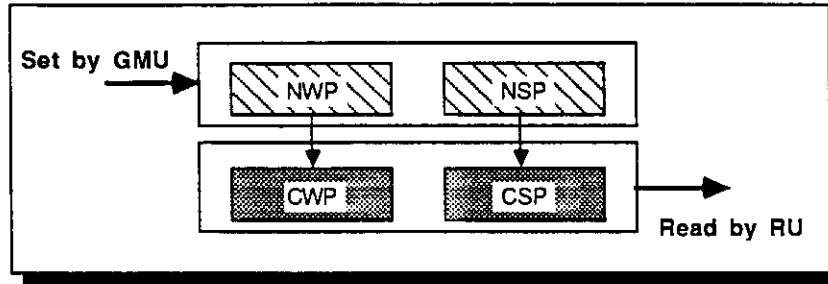


Figure 7.5: RU-GMU Interface via Goal Window Pointers

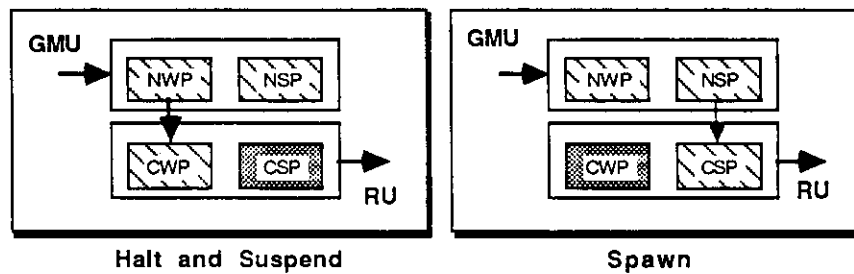


Figure 7.6: RU Interpretation of GMU Instructions

the management of the window pointer registers for the three goal management instructions: halt, spawn and suspend. The commit instruction does not affect the goal window pointers.

In Figure 7.6a, the interpretation of the halt instruction is shown to consist of moving NWP to CWP. The value of the CSP is not affected during the halt instruction. The same is true for the suspend operation. In case of the spawn instruction, NSP is moved to CSP whereas the value of the CWP is not changed, as shown in Figure 7.6. The new values of NWP and NSP are determined and set by GMU.

The NWP is used by the Instruction Unit to prefetch instructions from the next goal which are then used in the case of the halt and suspend instructions. If IU prefetches the first few instructions of the next goal using NWP, then the effective execution time of goal management instructions could be equal to zero. The effective execution time is then determined by the possible wait time caused by the single instruction goal management protocol, which does not allow the queuing of GMU instructions. We evaluate the GMU wait time as well as other

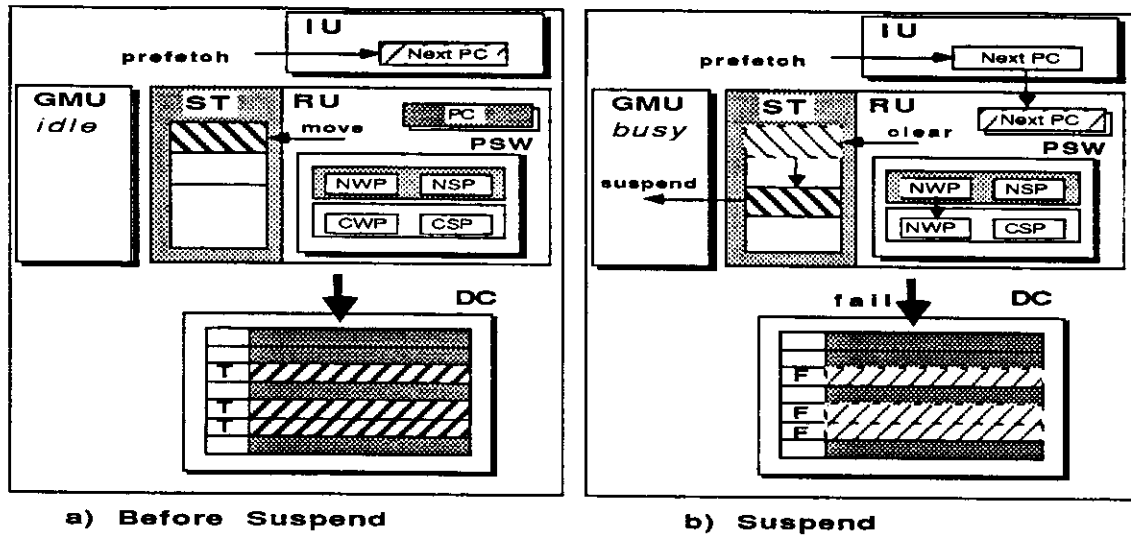


Figure 7.7: FCP Processor Execution of Overlapped Goal Suspension

goal management parameters in Chapter 8.

7.2.2 Goal Suspension and Activation: A Global View

Before we proceed to describe the execution of goal management operations using the Goal Cache, it is useful at this point to describe the global picture of the FCP processor execution, while GMU executes overlapped goal management operations and RU continues to perform goal reduction. In Figure 7.7 we show GMU, RU, DU and IU execution units as well as the Data Cache. The Goal Cache access is described in the following section. We now describe first the execution of goal suspension followed by goal activation.

Prior to the suspend instruction, ST contains values that GMU needs to fully implement goal suspension of the currently active goal stored in the Goal Cache. These values are pointers to suspension notes. The same suspension note pointers are also stored in the suspension variable locations in the Data Memory. If the suspension variable already contained a suspension note pointer, then this pointer is stored in ST. This case occurs if the current goal suspends on a variable that another goal previously suspended on. If the current goal suspends on a variable for the first time, then the variable location does not contain a suspension note pointer. In this case, the SN register value is store in the variable location, and

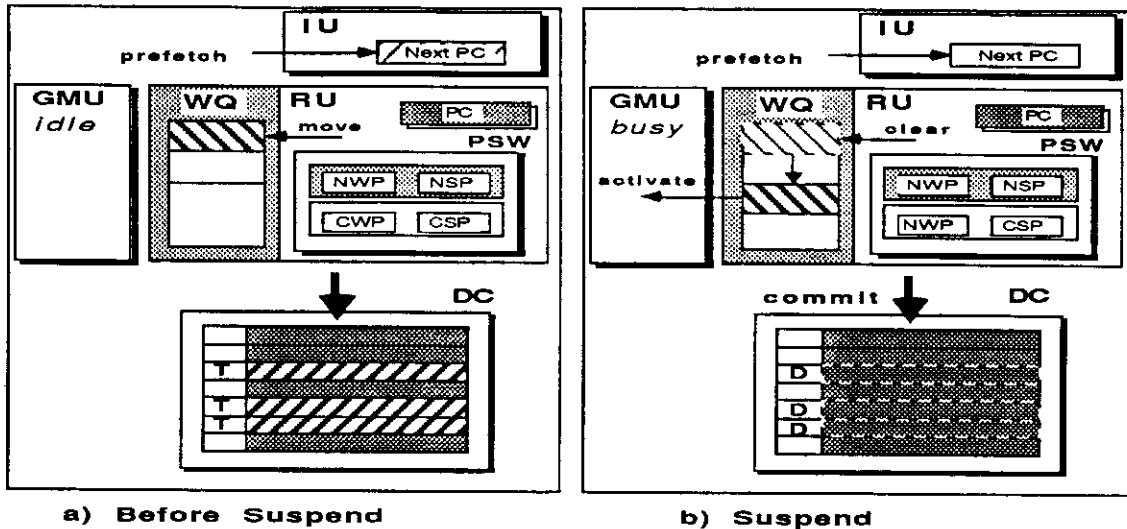


Figure 7.8: FCP Processor Execution of Overlapped Goal Activation

also in ST. The SN register value is set by GMU so that it always points to a preallocated suspension note. This value is determined by either incrementing the Goal Memory heap pointer, or by using the free list pointer SFL.

In Figure 7.7, we show the active goal which is denoted by CWP. GMU precomputed the values of the next window pointers NWP and NSP. These are shown above the pointers CWP and CSP. We also show that IU has prefetched the first instruction of the next goal. We label this instruction *next_PC*. In the Data Cache, during the clause-try, several assignments to the Data Memory are marked *trailed*.

When the suspend instruction is fetched and decoded, the following operations take place. The context switch to the next goal consists of moving the NWP to the CWP, and by shifting the contents of ST so that it is accessible by GMU. The first instruction of the next ready goal is given to RU to execute, while in the Data Cache, the status bits of the trailed values is changed to *free*.

With the above described steps, GMU is now completely decoupled from RU, and can perform overlapped goal suspension. In addition, GMU precomputes the next goal window pointers, and allows IU to prefetch the first instruction of the next goal.

A scenario similar to goal suspension, applies to the activation of goals at goal

commit time. Before the commit operation, WQ contains pointers to suspension records that were stored in the suspension variables in the Data Memory. The pointers are stored in WQ using move or load instructions, with WQ as the destination address.

When the commit operation is decoded, all of the trailed elements in the Data Cache are changed to *dirty*. Moreover, WQ is shifted so that a new WQ is used during continued goal reduction. During the clause-try that succeeds, RU may have place entries into ST as well as WQ. Therefore, at commit-time it is necessary to reset the ST stack pointer value, that is, clear ST. The implementation of the commit goal management instruction is shown in Figure 7.8.

7.3 GMU Instruction Execution Using the Goal Cache

The goal cache is primarily proposed for the efficient execution of GMU instructions. That is, GMU instructions could also be implemented without the use of a goal cache, but they would be less efficient. Most goal management instructions execute by simply manipulating the goal window status bits (GSB). However, in case of GC *overflow* or *underflow*, goals are either enqueued or prefetched to or from the goal memory. We now describe in detail the execution of GMU instructions using the goal cache. A variety of strategies are described but not evaluated. We consider a subset of a range of implementation approaches.

The GMU instruction execution algorithms described in the following sections are determined by the state of following three control primitives:

- Goal Cache Counter, (GCC). GCC keeps count of the number of ready goals in GC. The counter enables the detection of cases when GC contains only one goal (*underflow*) or has only one empty window (*overflow*). We define the counter increment and decrement operations labeled GCC+ and GCC- respectively. Note that if GC contains N goal windows, *overflow* is detected when ($GCC = N - 1$).
- Goal Memory Flag, (GMF). GMF maintains the status of the active queue in the goal memory. It is set if there is at least one goal in the active queue and reset otherwise. The active goal queue in GM is empty when the GQF = GQB = null. Two operations: goal prefetch and goal enqueue modify the GMF status after they manipulate the active goal queue.

- GMU Busy Flag, (GMUBusy). GMUBusy is set when GMU is performing goal management, that is, when GMU is busy. It is used by IU to schedule instructions. IU also sets the flag when GMU receives a new goal management instruction. The flag is reset by GMU upon its completion of the goal management instruction.

Let us further define the following multiple assignment statement used to denote changes to GSB performed by GMU: $GSB(w_1, \dots, w_N) = (s_1, \dots, s_N)$. That is, the status bit of goal window w_i is changed to s_i , for $1 \leq i \leq N$. Given the above defined GMU operations, we now specify the goal termination and spawning algorithms, followed by the goal suspension and activation algorithms. We use the following notation to denote that there is no data dependency between operations p_1, \dots, p_n : $[p_1 \parallel p_2, \dots, \parallel p_n]$. That is, these operations could execute concurrently.

7.3.1 Goal Termination

Goal termination is always performed in the goal cache. That is, the active goal is always located in one of the goal windows denoted by the CWP window pointer. The next goal to be scheduled for reduction, upon the termination of the current goal, is denoted by the NWP window pointer. The goal termination algorithm is shown in Figure 7.9.

When there is no cache underflow due to a halt instruction, the above algorithm shows that the active goal is marked as *free* and the next ready goal is marked *active*. GMU then computes the next ready goal and sets the NWP window pointer value. Upon completion of the halt instruction, the *GMUBusy* flag is reset.

In Figure 7.10 we depict the halt operation in GC, for the case that leads to a goal cache underflow. The function *next(ready)* returns the next *ready* window pointer value determined by inspection of the GSB. This may be the first available *ready* window in some priority order or perhaps in a round-robin order. In any case, for small sizes of N (number of goal windows) the next function may be implemented efficiently using combinational logic.

In case of *underflow*, goal termination prefetches a new goal from the non-empty active goal queue, and stores it in the (just) vacated goal window. Since it is not important where the goal is stored in GC, for simplicity it is stored in

```

GCC-.
IF (Not underflow):
    GSB(CWP,NWP) = (Free,Active).
    NWP = next(Ready).
IF (underflow) And (GMF):
    [ GSB(CWP,NWP) = (Ready,Active). ||
    NWP = CWP. ||
    GC(CWP) = Prefetch. ||
    GCC+].
IF underflow And Not GMF:
    GSB(CWP,NWP) = (Free,Active).
Reset(GMUBusy).

```

Figure 7.9: Goal Termination Algorithm

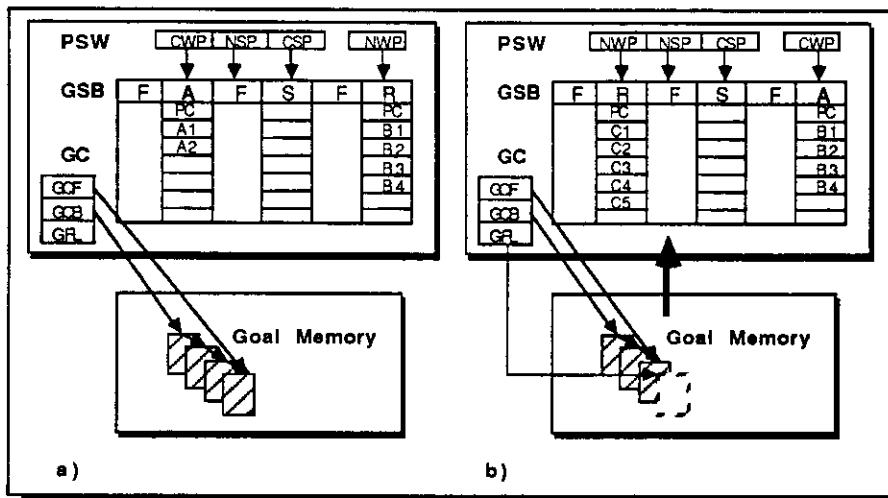


Figure 7.10: Goal Termination that Results in GC Underflow

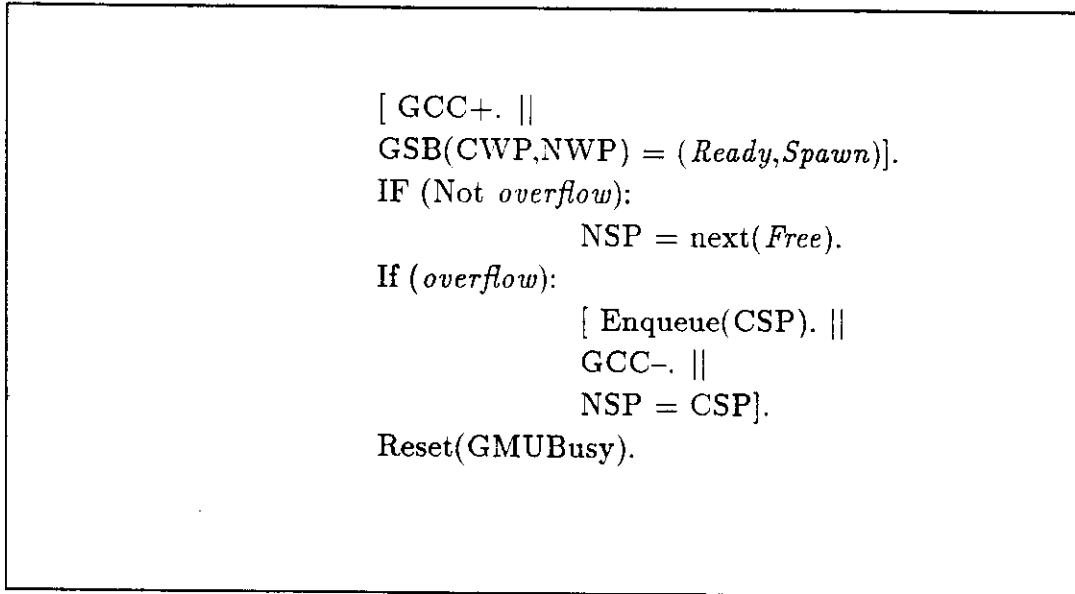


Figure 7.11: Goal Spawning Algorithm

the same window as the terminated goal.

7.3.2 Goal Spawning

Spawning a new goal by the currently active goal is always performed in GC using the current spawn window denoted by CSP. Therefore, the goal cache policy is to always maintain one empty goal window used for fast spawning. This window is marked as *spawn* in GSB. After placing the new goal argument pointers in CSW registers, spawning the goal is performed according to the algorithm described in Figure 7.11.

A spawn instruction that does not result in overflow is implemented by marking the currently spawned goal as *ready*, and the next empty window as *spawn*. GMU then computes the next free location and sets the NSP window pointer value.

Spawning a goal that results in the goal cache *overflow* is shown in Figure 7.12. As in the goal termination algorithm, the *next(Status)* function is used to find the next *free* window that will be designated as the next spawn window. In case of goal cache *overflow*, any goal window could be vacated from the goal cache and used for spawning the next goal. For simplicity, the above algorithm specifies that the most recently spawned goal is moved to the goal memory. An

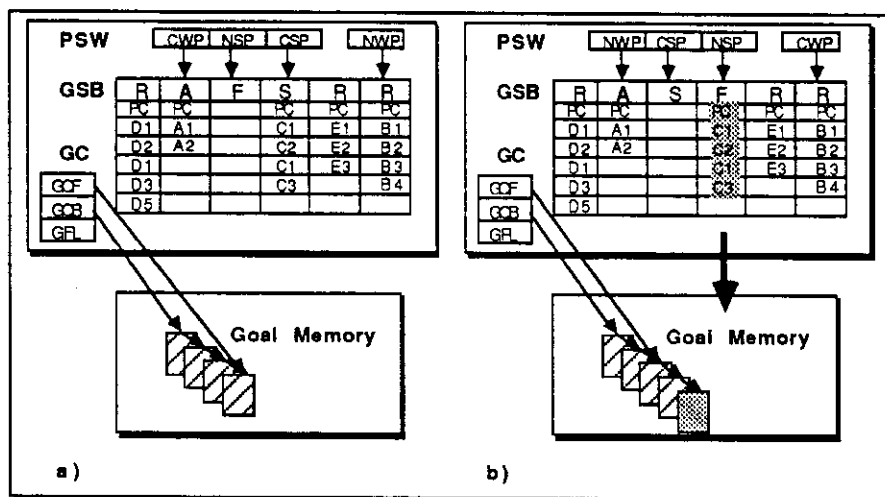


Figure 7.12: Goal Spawning that Results in GC Overflow

alternative is to move the nearest (first) *ready* goal rather than the most recently spawned goal.

7.3.3 Goal Suspension

GMU implements goal suspension by accessing the suspension record pointers stored in ST. The goal suspension control data structures used are the same as those described in Chapter 4. We assume that RU has switched to a new ST, and thus GMU has exclusive access to the ST that contains the suspension note pointers. The goal suspension algorithm performed by GMU is presented in Figure 7.13.

When the active goal suspends, it is moved from the goal cache into the goal memory, if this does not result in a goal cache *underflow* condition. If it does, and the active queue in memory is empty, the goal remains in the goal cache and is marked as *ready*. In case the active queue is not empty, the active goal is suspended by moving it from the goal cache to goal memory, and a new goal is prefetched.

If the underflow condition does not occur, the main part of the goal suspension algorithm is performed. This case is depicted in Figure 7.14. Let us assume that the current goal needs to suspend, waiting for three variable values to receive

```

GCC-.
IF (Not underflow):
    GSB(CWP,NWP) = (Free, Active).
    [ NWP = next(Ready). ||
    g = Allocate(goal_record). ||
    h = Allocate(hanger).]
    [ GM(h) = g. ||
    GM(h+) = 1. ||
    GM(g) = Store_goal(CWP). ]
    FOR EACH p in ST:
        [ GM(h+)+. ||
        value = GM(p). ]
        IF (value == null)
            THEN
                GM(p) = g.
            ELSE
                sn = Allocate(suspension_note).
                [ GM(sn) = g. ||
                GM(sn+) = GM(p+). ||
                GM(p+) = sn. ]
IF (underflow And Not GMF):
    GSB(CWP) = (Ready).
    NWP = CWP.
Reset(GMUBusy).

```

Figure 7.13: Goal Suspension Algorithm

assignments. The values stored in ST are denoted as p_1 , p_2 and p_3 and they are the same values stored in the suspension variables located in the Data Memory. The values are pointers to suspension notes stored in the Goal Memory area. Two of the suspension note pointers in ST, p_1 and p_2 point to already existing suspension lists, whereas p_3 points to a preallocated suspension note. The existing suspension control data structures prior to executing the suspend instruction are shown in Figure 7.14a. In Figure 7.14b, we show the newly allocated data structures that implement the suspension of the suspended goal. Note that the *reference count* value stored in the goal hanger of the newly suspended goal is equal to 3. This is because there are three suspension notes that point to this hanger.

Alternative goal suspension strategies could be proposed, that consider the possibility of leaving suspended goals in the goal cache, by marking them either as *ready* or labeled as *suspended*. In the first case, the goal suspension algorithm would be of a busy-waiting type. We suspect that this would result in a larger number of suspensions, and thus reduce performance. In the second case, the goal suspension algorithm becomes significantly more complex. It requires that goal activation check whether the activated goal is in the goal cache or not. Moreover, the case where all goals in the cache are marked as *suspended* would have to be detected and resolved. Evaluating various alternative goal suspension algorithms is beyond the scope of this thesis and is left for future research.

7.3.4 Goal Activation

As described in Chapter 5, most of the times the commit instruction does not result in the activation of goals. That is, WQ is empty. We refer to these cases as a *goal cache hit*. Those commit instructions that result in the activation of suspended goals are referred to as a *goal cache miss*. Upon goal activation, goals are enqueued onto the active goal queue in GM. Only in case of GC underflow are activated goals prefetched into GC. The goal activation algorithm is specified in Figure 7.15.

In Figure 7.16, we show the case where the previously suspended goals shown in Figure 7.14 receive data assignments during the clause-try that resulted in a successful goal reduction. Let us assume that at commit time, WQ contains suspension note pointers p_1 , p_2 and p_3 . These pointers are successively accessed and the suspension notes are traversed. The goals are activated by placing them

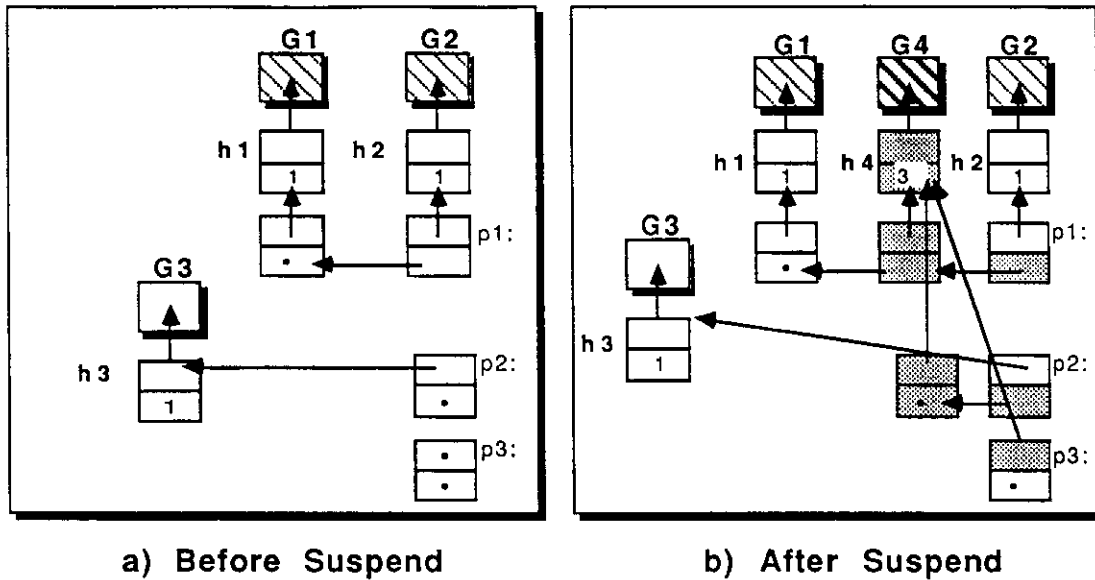


Figure 7.14: Goal Suspension Algorithm

onto the active queue, denoted using the GQF and GQB pointers. Upon activation, the suspension notes are garbage collected onto the Suspension Free List, denoted by SFL. Since there may be more suspension notes that point to the same hanger, a reference count is kept in the hanger location. The hanger is garbage collected only when the reference count of 0 is reached. When a goal is activated, the corresponding hanger is cleared so that later commits do not attempt to activate the same goal.

7.4 Examples of GMU Execution using the Goal Cache

We now show an example of GMU execution using the goal cache. Let us consider the following FCP program for quicksort. Each recursive call to the quicksort goal spawns four concurrent goals.

In Figure 7.18, we show the active goal window containing the quicksort goal, with the program counter labeled PC-Q and two argument pointers A_1 and A_2 . The arguments denote the input list of elements $[X|Xs]$ and the result variable *Sort*. The program is stored in the Instruction Memory (IM) whereas the list of input elements and the result are stored in the Data Memory (DM).

```

For EACH (p) in WQ:
    WHILE (GM(p) Not == null)
        h = GM(p).
        g = GM(h).
        IF (g Not == null)
            [ Activate(g). ||
              GM(h) = null. ]
        GM(h+)-.
        IF (GM(h+) == 0)
            Collect(h,SFL).
        p1 = GM(p+).
        Collect(p,SFL).
        p = p1.
Reset(GMUBusy).

```

Figure 7.15: Goal Activation Algorithm

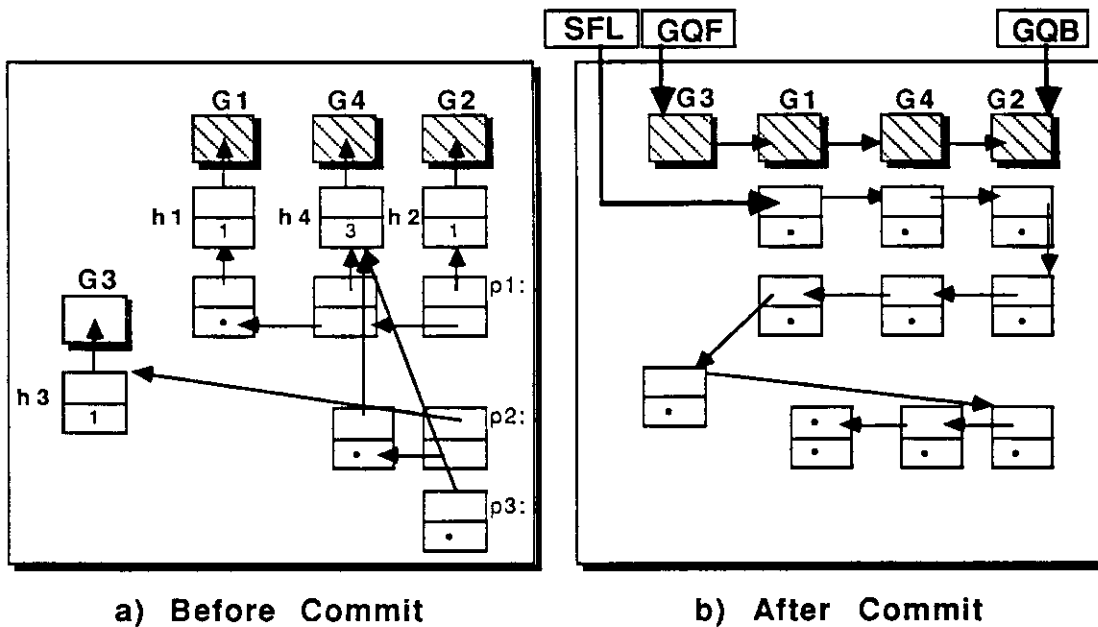


Figure 7.16: Goal Activation Algorithm

```

quicksort([X|Xs],Sort) :-
    partition(X?,Xs?,Small,Large),
    quicksort(Small?,S),
    quicksort(Large?,L),
    append(S?,[X|L?],Sort).
quicksort([],[]).

partition(X,[Y|Ys],[Y|Small],Large):-
    X <= Y | partition(X?,Ys?,Small,Large).
partition(X,[Y|Ys],Small,[Y|Large]):-
    X > Y | partition(X?,Ys?,Small,Large).
partition(X,[],[],[]).

```

Figure 7.17: Quicksort Program

In the PSW register, CGP points to the active goal window, CSP to the *spawn* window, NGP to the next *ready* goal and NSP points to a *free* window that will be used during the next spawning.

Instructions executed by RU first perform clause-head unification followed by clause-guard evaluation. Let us assume that this has completed successfully. At this point, four goals are spawned corresponding to two new quicksort goals, one partition goal and one append goal. Since a spawn window is already preallocated, spawning consists of moving argument pointers into the spawn window registers. As soon as the spawn operation is completed, RU continues executing the following instructions, thus creating the next goal. This is again performed by placing arguments into the newly preallocated spawn window.

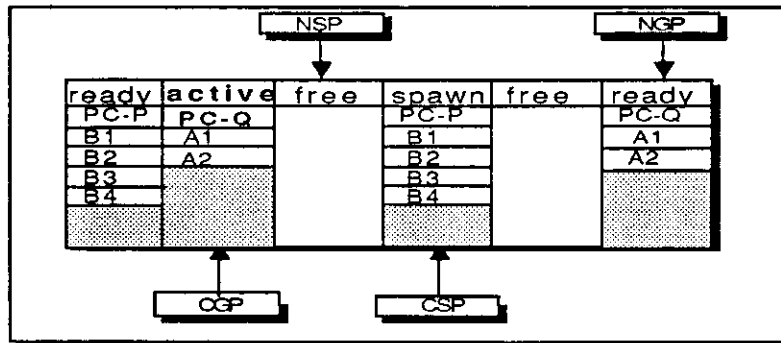
An optimization is performed in the case of the last goal. Instead of spawning it and then selecting a goal for scheduling, the active goal is modified and scheduled for execution. This procedure is referred to as *tail recursion optimization*.

Also shown in Figure 7.18 is the case when the last spawned goal results in goal cache overflow. We show that GMU vacates one goal cache window while RU still has one empty window for spawning.

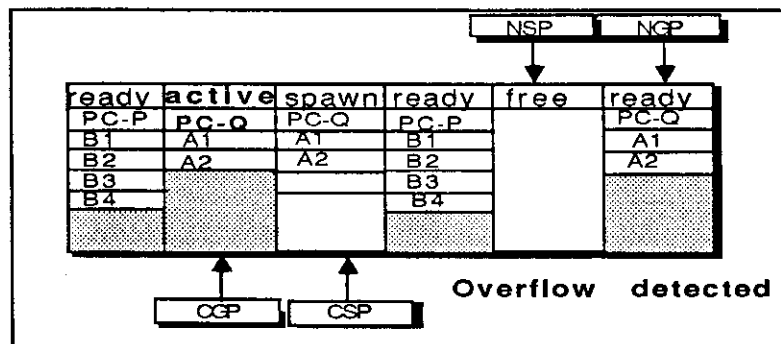
7.5 Properties of GMU Execution using GC

Overlapping goal management operations with goal reduction execution and using a goal cache for storing FCP goals has the following properties:

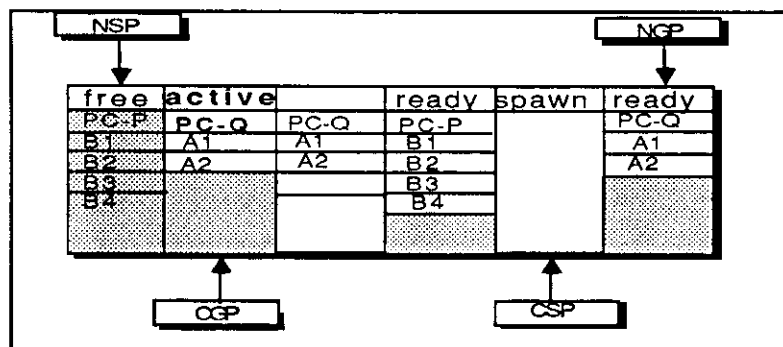
- Goals are always spawned in the CSW goal cache window. Without a goal cache, goals would either be created in memory, or first in registers and then moved to memory. Furthermore, by always having both the active and spawn goal in registers, an increase in the number of register-to-register operations, and thus performance is expected.
- GMU always maintains at least one empty goal window for spawning a new goal. Thus, RU never allocates memory for a goal record. This is always done by GMU, while the RU is performing goal reduction.
- Goal Memory is completely isolated from the rest of the processor architecture. That is, GMU privately accesses the GM and performs dynamic garbage collection. Allocating goals and goal control structures using the



a) Executing Quicksort Goal: Spawn Partition



b) Executing Quicksort Goal: Spawn Quicksort;



c) GMU Empties One Goal Window

Figure 7.18: Executing the Quicksort Program in the Goal Cache

same memory area results in frequent garbage collection interruptions, that degrade performance. This is in general a common problem of list-oriented languages. By dynamically garbage collecting in the Goal Memory we isolate the problem to the Data Memory area. Maintaining a separate control memory could also be used in the sequential abstract machine and may also result in less frequent garbage collection interruptions.

- FCP goals are generally light-weight computations. Goals spawned in the Goal Cache may terminate in the Goal Cache without ever being removed. Therefore, goal caching reduces the number and time spent allocating memory for goal records and other goal management control data structures.
- Part of the FCP programming style is to perform computations using communicating goals, often spawned in the same committed clause. The goal cache policy described in this section stores recently spawned goals closer to the processor in order to maximize performance. What is trading-off is goal scheduling fairness. Goals spawned within a short distance of each other are more likely to be found together in the goal cache, thus capturing inter-goal locality of communication.

CHAPTER 8

Analytic Performance Evaluation of RU-GMU Execution

In this chapter we analytically evaluate the performance of overlapped execution of goal management and goal reduction, as proposed in Chapter 6 and described in Chapter 7. First we specify the performance measures followed by the analytic performance model and finally the performance analysis for a range of characteristic parameter values.

The approach used for performance modeling in this chapter differs from that presented in Chapter 5 in the following way. Since we are considering here a specific physical machine architecture, the proposed model contains parameters that describe its organization and instruction execution as opposed to the more high-level model used in Chapter 5. As far as the system workload is concerned, we use the same set of benchmark programs and their corresponding parameter values.

8.1 RU-GMU Performance Measures

To evaluate RU-GMU performance we define the following four performance measures:

1. Average Instruction Execution Time ($\bar{\tau}$).
2. Average RU-GMU Wait Time (\bar{W}).
3. Relative Effective GMU Execution Time (R_e).
4. GMU Utilization (U_{gmu}) and RU Utilization (U_{ru}).

The average RU-GMU instruction execution time, $\bar{\tau}$, characterizes the performance of program execution for a given system workload. It corresponds to the average number of processor cycles required to execute a single processor instruction. For a specific processor instruction set the objective is to define an

execution model and (cost-effective) architectural support to reduce the average execution time. The average instruction execution time is not necessarily indicative of overall system performance which may depend on other system components.

The average RU-GMU wait time, \overline{W} , is a measure of the time that RU waits for GMU. The wait time directly affects program execution time and the objective is to reduce this overhead. As a performance measure, the average wait time is also included as part of the average instruction execution time and the relative effective GMU execution time discussed in the following paragraph. However, since we consider it an important performance measure on its own, we make a point of analyzing it separately and using this analysis to evaluate other performance measures.

The relative effective RU-GMU execution time, R_e , compares the effective times performing goal reduction in RU and goal management in GMU, for a given system workload. It is a measure of the RU-GMU execution model which includes inter-unit communication and synchronization. The objective is to reduce the effective time for goal management execution below a specified value $K\%$. In an effort to completely overlap goal reduction and goal management, the objective may be to set the value for K to be very small (for example $R_e < K = 3 - 4\%$).

The GMU utilization, U_{gmu} , determines the time spent performing goal management relative to the program execution time, for a given system workload. Similarly, RU utilization, U_{ru} , determines the time spent performing goal reduction relative to the program execution time. Together, the utilization factors are a measure of the workload balance. Ideally, for a balanced system, the two concurrent units RU and GMU are equally loaded resulting in 100% utilization. By investigating the workload balance between concurrently executing units one may investigate more cost-effective use of available resources. In general, this is one way of achieving a more efficient, balanced system without bottlenecks (see [Ferr78] for details). We now describe the RU-GMU system organization followed by the description of the performance model parameters.

8.2 System Organization and Parameters

In Figure 8.1 we show the RU-GMU system organization used to define the performance model. It shows the functional units RU, GMU as well as the goal cache (GC) and the goal memory. RU executes a RISC-type instruction set

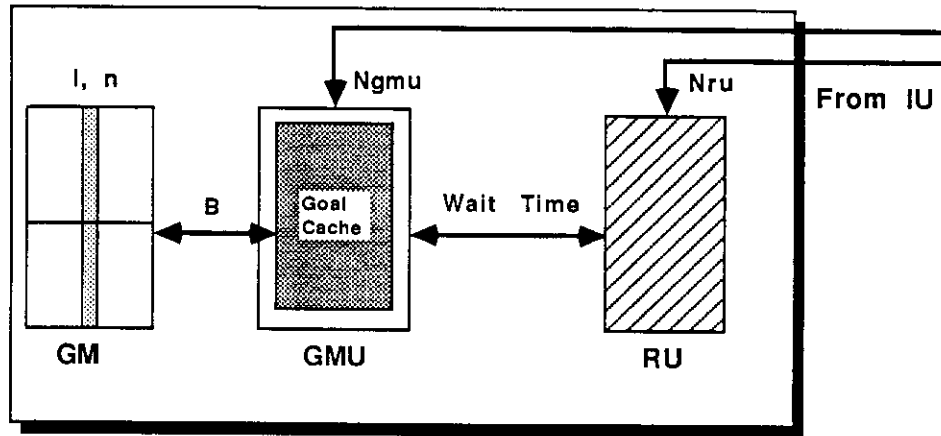


Figure 8.1: RU-GMU Performance Model System Organization

whereas GMU executes instructions: halt, spawn, suspend, and commit. GMU instructions that execute only in the goal cache are modeled as goal cache hits. GMU instructions that require access to the goal memory are considered a goal cache miss. In the implementation algorithm discussed in Chapter 7, halt and spawn result in a goal cache hit if there is no goal cache *underflow* or *overflow*. Commit instructions that result in the activation of previously suspended goals in the goal memory are considered a goal cache miss. However, a commit instruction that does not activate goals is considered a goal cache hit. If the suspend instruction results in the transfer of the current goal to goal memory, a goal cache miss results, otherwise a goal cache hit occurs.

We group the performance model parameters into the following three categories:

1. Implementation dependent parameters.
2. Workload parameters.
3. Architecture and workload dependent parameters.

Implementation Dependent Parameters

All the execution times of GMU and RU instructions are implementation dependent. These parameters are denoted with the lower case letter t . For ex-

ample, the average execution time of RU instructions is denoted as \bar{t}_{ru} . Similarly, the execution time of a GMU instruction i that results in a goal cache miss is denoted as t_i^m and the execution time of a goal cache hit as t_i^h .

Goal management operations are overlapped with goal reduction execution. The *effective* execution time of goal management operations is also implementation dependent. It depends on the overlapped execution algorithm and architectural support. For example, ideally, the effective execution time of a goal management operation is equal to zero. We denote the effective execution time of GMU instructions that result in a goal cache miss as $t_{gmu}^{e,m}$ and the effective execution time of GMU instructions that result in a goal cache hit as $t_{gmu}^{e,h}$.

Another implementation dependent parameter is the goal memory bandwidth denoted as B . Let D denote the data width of the GMU parallel memory port to goal memory, n the ratio of GMU processor cycle time and goal memory access time, p the GMU clock cycle time and I the interleaved factor of the goal memory system. The goal memory bandwidth B is then expressed as:

$$B = \frac{I \times D}{n \times p} \quad (8.1)$$

The degree of memory interleaving, I , of the goal memory system determines the maximum number of memory requests that may be serviced concurrently by separate goal memory modules.

Workload Parameters

In Chapter 5 we described a set of workload parameters that characterize the execution of the selected FCP benchmark programs. The same set of parameters is used in this chapter. For example, the average number of variables a goal suspends on, N_{var}^s , the average number of goals activated at clause commit, N_{act}^c , and the average size of a goal, S are also used. In addition, N_{gmu} denotes the total number of GMU instructions executed and F_{halt} , F_{sp} , F_{susp} and F_{com} the fraction of executed goal management instructions halt, spawn, suspend and commit respectively.

Architecture and Workload Dependent Parameters

Some parameters in the analytic model are both architecture and workload dependent. By architecture we mean both hardware support and implementation

aspects such as the state of the art of compiler technology. Moreover, these parameters also depend on features of the workload, such as locality of memory referencing.

For example, N_{ru} denotes the number of executed RU instructions and N_{gmu} the number of GMU instructions. Their frequencies are labeled F_{ru} and F_{gmu} respectively. F_{gmu}^h denotes the goal cache hit ratio, that is, the fraction of all goal management instructions that do not result in goal memory access. We denote the hit ratio of each goal management instruction i as F_i^h .

8.3 Performance Parameter Measurement

In Chapter 5 we showed how the workload parameters are obtained by performing analysis at the abstract machine level. The analysis produces results that are independent of the host machine implementation and abstract machine emulation language. Moreover, it was argued that lower-level, machine-dependent analysis is appropriate only if the specific machine implementation is being optimized and the results can not be simply applied to other machine implementations.

After specifying the special-purpose processor architecture for FCP, we are able to determine the machine dependent parameter values such as the execution time of machine instructions. For example, the execution rate of a RISC instruction set may be approximated as one instruction per processor cycle.

However, those performance model parameters that depend on both the architecture and implementation have to be determined using simulations unless the processor architecture is actually manufactured. We use the following approach. The abstract machine execution described in Chapter 5 is compiler based and executes a set of abstract machine instructions. Each one of the abstract machine instructions is implemented using the specific instruction set of the FCP processor. The instructions are dynamically counted and approximate the program execution on the special-purpose FCP processor.

The above described approach is an approximation because of the following features:

- The abstract machine goal scheduling strategy differs from what is proposed as part of the FCP processor execution model.

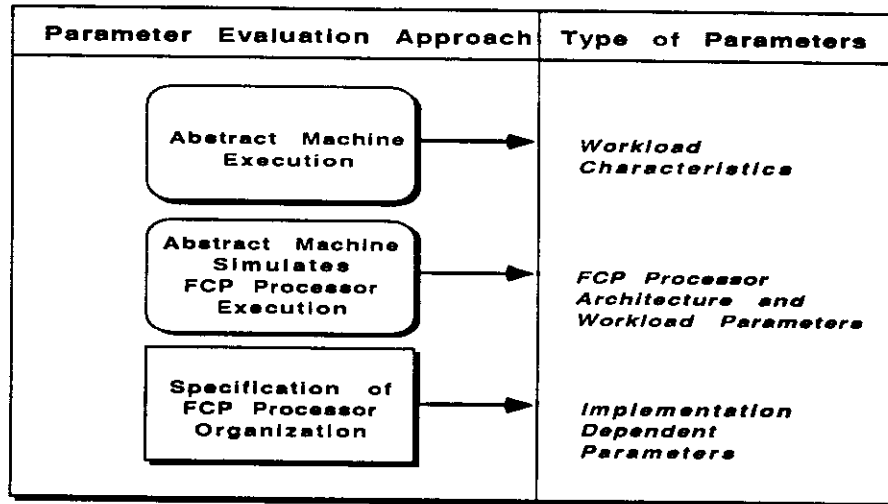


Figure 8.2: Performance Parameter Measurement Approach

- The simulated FCP processor instruction execution uses the current version of the abstract machine compiler. More efficient compiler techniques are thus not modeled. In this sense, the analysis will result in more time consuming clause-tries, as it was discussed in Chapter 5. Improvements in compiler technology and particularly advances in clause indexing strategies are expected to reduce the clause-try time and thus increase the relative time that goal management is being performed.

The first issue does not affect the obtained results for the following reason. The programs that are used are very large applications and therefore correspond to a reasonable average of program behavior. From this point of view, the obtained trace can be considered as a good average that is characteristic of the considered *system's development workload*.

The second issue, that is the compiler technology, does affect the obtained measurements in the following way. We expect that a lower bound on the relative execution time of goal management is evaluated. In other words, improvements in compilation techniques may reduce the clause-try time and thus increase the relative execution time of goal management.

8.4 Performance Model

We now express the four performance measures using the previously defined parameters.

8.4.1 Average Instruction Execution Time

The average instruction execution time, $\bar{\tau}$, is equal to the sum of the average execution time of RU instructions and the average effective execution time of GMU instructions. That is,

$$\bar{\tau} = (1 - F_{gmu})\bar{t}_{ru} + F_{gmu}\bar{t}_{gmu}^e \quad (8.2)$$

The effective execution time of GMU instructions, \bar{t}_{gmu}^e , consists of two parts. First, each GMU instruction, whether it leads to a goal cache hit or miss, results in an effective execution time of $t_{gmu}^{e,h}$ and second, an average wait time of \bar{W}_{gmu} is incurred by each executed GMU instruction. If we label the total wait time as W then the average wait time per GMU instruction is expressed as $\bar{W}_{gmu} = \frac{W}{N_{gmu}}$. The GMU effective execution time is equal to:

$$\bar{t}_{gmu}^e = t_{gmu}^{e,h} + \bar{W}_{gmu} \quad (8.3)$$

In other words, $t_{gmu}^{e,h}$ denotes the minimum effective execution time of a goal management instruction. It reflects the time it takes for RU to interpret the instruction while GMU performs the overlapped execution. The effective execution time of a GMU instruction hit depends on the architectural support such as instruction prefetching. This will be further discussed later in this chapter.

If we replace expression (8.3) in equation (8.2) we derive the following expression for the average instruction execution time $\bar{\tau}$:

$$\bar{\tau} = (1 - F_{gmu})\bar{t}_{ru} + F_{gmu}t_{gmu}^{e,h} + \bar{W} \quad (8.4)$$

where \bar{W} denotes the average RU-GMU wait time per total number of executed instructions. That is, $F_{gmu} \times \bar{W}_{gmu} = \bar{W}$.

From expression (8.4), one can see that the average instruction execution time $\bar{\tau}$ consists of three parts, represented as:

$$\bar{\tau} = \bar{\tau}_1 + \bar{\tau}_2 + \bar{\tau}_3 \quad (8.5)$$

The first part, $\bar{\tau}_1$, represents the average instruction execution time of RU instructions, $\bar{\tau}_2$ represents the average effective time spent by RU interpreting GMU instructions regardless of whether a goal cache hit or miss occurred, and $\bar{\tau}_3$ represents the average wait time per executed instruction, \bar{W} . One can note that if program execution does not contain GMU instructions, then the average instruction execution time $\bar{\tau}$ is equal to the average instruction execution time of RU instructions \bar{t}_{ru} , since the average wait time is then equal to zero.

8.4.2 Average RU-GMU Wait Time, \bar{W}

The average RU-GMU wait time, \bar{W} , is equal to the ratio of the total RU-GMU wait time W and the number of executed GMU and RU instructions N . That is,

$$\bar{W} = \frac{W}{N} = F_{gmu} \times \bar{W}_{gmu} \quad (8.6)$$

To determine the RU-GMU Wait Time, W , we consider the execution of two consecutive GMU instructions, as shown in Figure 8.3. Let Δ^i denote the elapsed time between two consecutive GMU instructions gmu_i and gmu_{i+1} , and t_{gmu}^i the time it takes to execute GMU instruction i . If the duration of GMU instruction i is less than the distance Δ^i , the wait time, w_i , is equal to zero. However, if the duration is greater than the distance, a non-zero wait time is incurred. Thus, we express the wait time for the i^{th} GMU instruction as:

$$w_i = \begin{cases} (t_{gmu}^i - \Delta^i) & \text{if } \Delta^i < t_{gmu}^i \\ 0 & \text{otherwise} \end{cases} \quad (8.7)$$

The total RU-GMU wait time is then represented as the sum over all executed GMU instructions:

$$W = \sum_{i=1}^{i=N_{gmu}} (w_i) \quad (8.8)$$

Using expressions (8.7) and (8.8) to evaluate the RU-GMU wait time requires that one compute the execution time of each GMU instruction, and the distance between consecutive instructions. As a performance model, this does not seem practical, since one must consider too many combinations of *execution times versus distance* distributions. However, the performance model can be simplified due to the following features:

1. There are only four GMU instructions.

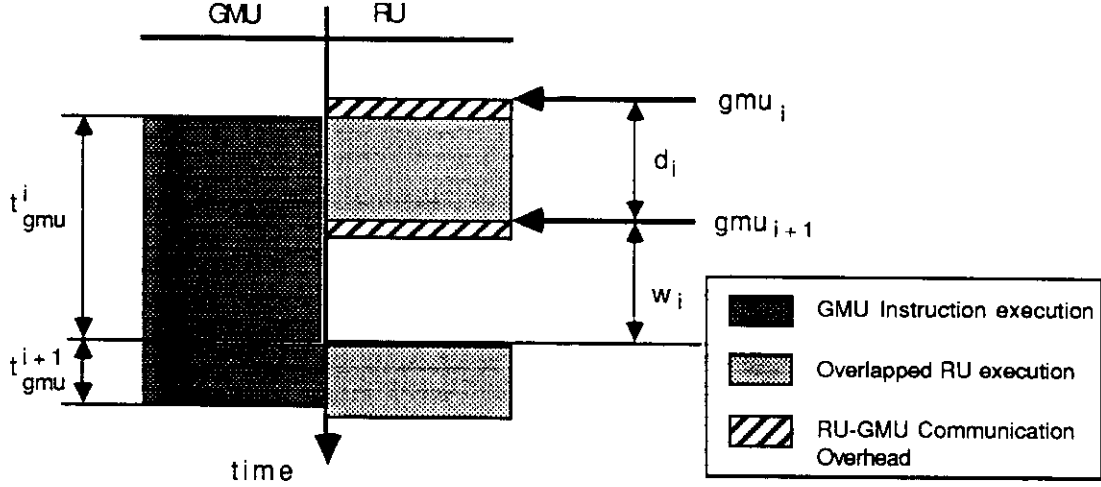


Figure 8.3: RU-GMU Instruction Wait Time w_i

2. The GMU instruction execution times are well defined.
3. The number of significant program and implementation parameters in the defined performance model is small.

The RU-GMU wait time can be divided according to the four GMU instructions as follows:

$$W = W_{halt} + W_{sp} + W_{susp} + W_{com} \quad (8.9)$$

Moreover, as it was described in Chapter 5, goals in FCP programs typically suspend on several variables, with the average being $N_{var}^s = 2.6$ and the number of goal activations is typically 1 or 2, with the average being $N_{act}^c = 1.12$. Therefore, the number of significant program parameter values required to model the execution time of suspend and commit is small. We describe the execution times of GMU instructions in more detail later in this chapter.

From expression 8.7, one can see that the meaningful values for inter-GMU instruction distances are those that are less than the maximum value of the GMU instruction execution times. All GMU instruction distances beyond this value can be ignored, since they can not interrupt the overlapped GMU execution. Let t_{gmu}^{max} denote the longest execution time for a single GMU instruction. Let i denote the inter-goal instruction distance. and f_j^i the frequency with which the GMU instruction $j \in (halt, spawn, suspend, commit)$ is interrupted by another GMU instruction at distance i . For example, f_{halt}^6 denotes the frequency of interruptions of the halt instruction at distance $i = 6$.

For the suspend and commit instructions, the execution time depends on the number of suspension variables N_{var}^s and the number of activated goals N_{act}^c respectively. The frequency with which a suspend instruction that suspends on N_{var}^s variables (and a commit instruction that activates N_{act}^c goals) is interrupted at instruction distance i is denoted as $f_{suspend(N_{var}^s)}^i$ (and similarly $f_{commit(N_{act}^c)}^i$).

If we define a *unary* function $u(x)$ as :

$$u(x) = \begin{cases} x & x \geq 0 \\ 0 & \text{otherwise} \end{cases} \quad (8.10)$$

then the total wait time, W , is expressed as:

$$\begin{aligned} W &= \sum_i^{t_{gmu}^{max}} [u(t_h - i)f_h^i + u(t_{sp} - i)f_{sp}^i \\ &+ \sum_{N_{var}^s=1}^{N_{var}^{s,max}} (u(N_{var}^s t_{suspend} - i)f_{suspend(N_{var}^s)}^i) \\ &+ \sum_{N_{act}^c=0}^{N_{act}^{c,max}} (u(N_{act}^c t_{commit} - i)f_{commit(N_{act}^c)}^i)] \end{aligned} \quad (8.11)$$

where $N_{var}^{s,max}$ denotes the maximum number of variables a goal suspends on, and $N_{act}^{c,max}$ denotes the maximum number of activated goals at clause-commit.

8.4.3 Relative Effective GMU Execution Time, R_e

The relative effective GMU execution time compares the total effective GMU execution time T_{gmu}^e with the absolute RU execution time T_{ru} . That is:

$$R_e = \frac{T_{gmu}^e}{T_{ru}} = \frac{\bar{t}_{gmu}^e \times N_{gmu}}{\bar{t}_{ru} \times N_{ru}} \quad (8.12)$$

If we divide the above equation with the total number of executed instructions N we derive the following expression for the relative effective GMU execution time:

$$R_e = \frac{\bar{t}_{gmu}^e}{\bar{t}_{ru}} \times \frac{F_{gmu}}{F_{ru}} = \frac{\bar{t}_{gmu}^e}{\bar{t}_{ru}} \times \frac{F_{gmu}}{1 - F_{gmu}} \quad (8.13)$$

Using expression (8.3) for the effective GMU instruction execution time in the above expression, we derive the following expression for R_e :

$$R_e = \frac{t_{gmu}^{e,h} + \bar{W}_{gmu}}{\bar{t}_{ru}} \times \frac{F_{gmu}}{1 - F_{gmu}} \quad (8.14)$$

One should note that another way to represent the relative effective GMU execution time R_e is using the notation for the three parts of the average instruction execution time, $\bar{\tau}$ found in expression (8.5). That is,

$$R_e = \frac{\bar{\tau}_2 + \bar{\tau}_3}{\bar{\tau}_1} \quad (8.15)$$

8.4.4 GMU and RU Utilizations, U_{gmu} , U_{ru}

We define the GMU utilization U_{gmu} , as the ratio of the GMU execution time, T_{gmu} , and the total program execution time, T .

$$U_{gmu} = \frac{T_{gmu}}{T} = \frac{N_{gmu} \times \bar{t}_{gmu}}{N \times \bar{\tau}} \quad (8.16)$$

That is,

$$U_{gmu} = F_{gmu} \times \frac{\bar{t}_{gmu}}{\bar{\tau}} \quad (8.17)$$

where \bar{t}_{gmu} represents the average GMU execution time defined in (8.21), and $\bar{\tau}$ is defined by expression (8.4).

The RU utilization U_{ru} is similarly defined as the ratio of the RU execution time, T_{ru} , and the total program execution time T . That is,

$$U_{ru} = \frac{T_{ru}}{T} \quad (8.18)$$

If we use the notation of expression (8.5), we express RU utilization as:

$$U_{ru} = \frac{\bar{\tau}_1}{\bar{\tau}_1 + \bar{\tau}_2 + \bar{\tau}_3} \quad (8.19)$$

If we divide the above expression with $\bar{\tau}_1$ and use equation (8.15), we derive the following expression for the RU utilization:

$$U_{ru} = \frac{1}{1 + R_e} \quad (8.20)$$

We now evaluate in more detail the average GMU instruction execution time \bar{t}_{gmu} .

8.4.4.1 Average GMU Instruction Execution Time, \bar{t}_{gmu}

To evaluate the GMU utilization and the RU-GMU wait times, we evaluate the GMU absolute instruction execution times as follows. Let GMU execute instructions with the average execution times: \bar{t}_{halt} , \bar{t}_{sp} , \bar{t}_{susp} , \bar{t}_{com} and with instruction frequencies: F_{halt} , F_{sp} , F_{susp} , and F_{com} . The average absolute GMU instruction execution time \bar{t}_{gmu}^a is thus expressed as:

$$\bar{t}_{gmu} = F_{halt}\bar{t}_{halt} + F_{sp}\bar{t}_{sp} + F_{susp}\bar{t}_{susp} + F_{com}\bar{t}_{com} \quad (8.21)$$

Moreover, if we denote the GMU instruction execution times for a goal cache hit and miss as t_{gmu}^h , t_{gmu}^m the average execution times for the halt, spawn, suspend and commit instructions are expressed as:

$$\begin{aligned}
\bar{t}_{halt} &= F_{halt}^h t_{halt}^h + (1 - F_{halt}^h) t_{halt}^m \\
\bar{t}_{sp} &= F_{sp}^h t_{sp}^h + (1 - F_{sp}^h) t_{sp}^m \\
\bar{t}_{susp} &= F_{susp}^h t_{susp}^h + (1 - F_{susp}^h) t_{susp}^m \\
\bar{t}_{com} &= F_{com}^h t_{com}^h + (1 - F_{com}^h) t_{com}^m
\end{aligned} \tag{8.22}$$

As described in Chapter 7, the execution of GMU instructions that result in a goal cache hit is efficiently implemented by manipulating the goal window status bits. However, the execution time of GMU instructions that result in a goal cache miss require access to goal memory, and the manipulation of goal memory control structures.

The frequency of goal cache misses that result during the halt and spawn instruction correspond to the goal cache *underflow* and *overflow* respectively. In each case S goal memory words, corresponding to the goal size, are transferred to/from memory. In addition, the garbage collection of discarded data structures is performed where necessary.

If we assume that a goal memory word consists of 4 bytes, that the goal size is S words, the goal memory bandwidth B bytes/cycle, and a single memory access time is equal to n cycles, the execution time of a halt or spawn instruction that result in a goal cache miss is represented as:

$$\begin{aligned}
t_{halt/sp}^m &= \lceil 4 \times (S + 3) / B \rceil && ; \text{consecutive memory r/w accesses} \\
&+ \lceil 4 / B \rceil && ; \text{1 memory access} \\
&+ 1 && ; \text{1 processor cycle}
\end{aligned} \tag{8.23}$$

The execution time of the suspend instruction that results in a goal cache miss is linearly proportional to the number of variables, N_{var}^s , the current goal suspends on. Similarly, the execution time of the commit operation is linearly proportional to the number of goals activated at commit time, N_{act}^c . The suspend instruction always results in the transfer of the suspended goal to the goal memory, thus effectively behaving as a goal cache miss. Besides the transfer of $4 \times S$ goal memory bytes, suspension control data structures are allocated and manipulated as described in Chapter 4. The execution time for goal suspension

is then represented as:

$$\begin{aligned}
t_{susp}^m &= \quad [4 \times (S + 2)/B] && ; \text{ move goal to memory} \\
&+ \quad 2(\lceil(2/B)\rceil) && ; \text{ allocate hanger} \\
&+ \quad [N_{var}^s(3 \times \lceil 4/B \rceil + \lceil(2/B)\rceil + 2)] && ; \text{ allocate suspension notes}
\end{aligned} \tag{8.24}$$

Note that since all suspend instructions result in a goal cache miss, $F_{susp}^h = 0$.

The commit instruction requires that goals be placed onto the active goal queue and that the data structures used during goal suspension be garbage collected. The commit instruction execution time for a goal cache miss is expressed as:

$$\begin{aligned}
t_{com}^m &= [N_{act}^c(3\lceil(2/B)\rceil + \lceil 4/B \rceil)] && ; \text{ activate goals} \\
&+ \quad 4 && ;
\end{aligned} \tag{8.25}$$

In summary, we characterize the execution GMU instructions as follows:

- All GMU instructions that result in a goal cache hit are efficiently implemented by manipulating the goal window status bits.
- GMU instructions that result in a goal cache miss are memory bound. Therefore, their execution time is proportional to the goal memory bandwidth.
- All suspend instructions result in a goal cache miss.

8.5 Performance Model Parameter Values

We now present the measured values for each of the three groups of performance model parameters described earlier, in Section 8.3.

Workload Parameter Values

The workload parameter values used in the analytic performance evaluation model are shown in Table 8.1, and are the same values shown in Chapter 5.

	Workload Parameters			Architecture and Workload Parameters		
	S	N_{var}^s	N_{act}^c	F_{gmu}	$F_{gmu}^h(4)$	$F_{gmu}^h(N)$
Ave.	5.8	2.6	1.12	3.6%	37%	70%

Table 8.1: Parameter Values

Architecture and Workload Parameter Values

In Table 8.1 we show the average number of executed RU and GMU instructions as well as the frequency of GMU instructions, F_{gmu} .

The goal cache hit ratio for the halt and spawn instructions depend on the goal cache size and the locality of goal management behavior. However, the goal cache miss ratio due to suspend and commit instructions is unaffected by the goal cache size, and depends only on the program characteristics. We consider two extreme cases for the goal cache size. First, the *minimum* goal cache which consists of only 4 goal windows: *active*, *spawn*, *free* and *ready*. The second case is when the goal cache is sufficiently large so that no cache *overflow* or *underflow* conditions occur. That is, the only instructions that result in a goal cache miss are all suspend and some commit operations. In Table 8.1 we show the average goal cache hit ratios for a minimum goal cache size (GC=4) and for a large goal cache (GC=N).

Implementation Parameter Values

The goal memory bandwidth required during program execution may be estimated using the following *back-of-the-envelope* calculation. If we assume that on the average, each GMU instruction results in the transfer of $4 \times (S + 1)$ bytes to goal memory, the required goal memory bandwidth is equal to:

$$B = \frac{N_{gmu} \times 4(S + 1)}{T} = \frac{F_{gmu} \times S}{\bar{\tau}} \quad (8.26)$$

where T denotes the total program execution time. For the average number of goal arguments of $S = 7$, $F_{gmu} = 4\%$ and $\bar{\tau} \approx 1$, the required goal memory bandwidth is $B = 1.3\text{bytes/cycle}$.

In the performance model analysis, we consider three different goal memory bandwidth values: $B = 2\text{bytes/cycle}$, 4bytes/cycle and 8bytes/cycle . The respective bandwidth is obtained by increasing the degree of goal memory interleaving, I , as discussed in (8.1).

	B=2bytes/cycle			B=4bytes/cycle			B=8bytes/cycle		
t_{gmu}^m	$t_{halt/sp}^m$	t_{susp}	t_{com}	$t_{halt/sp}^m$	t_{susp}	t_{com}	$t_{halt/sp}^m$	t_{susp}	t_{com}
cycles	19	42	10	10	26	9	6	22	9

Table 8.2: GMU Instruction Execution Times

In Table 8.2 we show the execution times of GMU instructions that result in a goal cache miss. The values are obtained by replacing the goal memory bandwidth value B and the average goal size S in expressions 8.23, 8.24 and 8.23.

8.6 Performance Model Analysis

We begin the performance analysis by first considering the objective to contain the relative effective execution time R_e below the value $K\%$. Using expression 8.14, the condition $R_e < K\%$ is expressed as follows:

$$R_e = \frac{F_{gmu} t_{gmu}^{e,h} + \overline{W}}{(1 - F_{gmu}) \bar{t}_{ru}} \leq K \quad (8.27)$$

and using expression 8.5, the average instruction execution time $\bar{\tau}$ is then expressed as:

$$\bar{\tau} \leq \bar{\tau}_1(1 + K) \quad (8.28)$$

To determine the necessary conditions for (8.27) to hold, let us assume that the average instruction execution rate of RU instructions is equal to one, ($\bar{t}_{ru} = 1$). That is, if the conditions are met when $t_{ru} = 1$, then the less restrictive cases when $t_{ru} > 1$ is also satisfied. Let us further consider the minimum value for R_e which results when the average wait time per instruction is equal to zero, ($\overline{W} = 0$). That is, let:

$$R_e^{min} = R_e|_{\overline{W}=0} \quad (8.29)$$

The relative effective GMU execution time is then expressed as:

$$R_e^{min} = \frac{F_{gmu} t_{gmu}^{e,h}}{(1 - F_{gmu})} \quad (8.30)$$

The above expression (8.30) for the relative effective GMU execution time represents the minimum possible value for the RU-GMU overhead which is achieved when the wait time is reduced to zero. The value for the effective execution time

of a GMU instruction that leads to a goal cache hit, $t_{gmu}^{e,h}$, is implementation dependent. We consider the following two cases. If $t_{gmu}^{e,h} = 0$, then the minimum value for R_e is also zero. If, however, $t_{gmu}^{e,h} = 1$, then the minimum relative effective GMU execution time is given with the following expression:

$$R_e^{min} = \frac{1}{\frac{1}{F_{gmu}} - 1} \quad (8.31)$$

For values of $F_{gmu} < 1$ that are small, we can approximate the relative effective GMU execution time as:

$$R_e^{min} \approx F_{gmu} \quad (8.32)$$

That is, the relative execution time is bound by the frequency of GMU instructions. Therefore, in the case of $F_{gmu} = 10\%$, $R_e^{min} > 10\%$.

In other words, if the frequency of GMU instructions is F_{gmu} , and the minimum effective execution time of GMU instructions is $t_{gmu}^{e,h} = 1$ processor cycle, and the average execution time of RU instructions is also $\bar{t}_{ru} = 1$ processor cycle, then $R_e^{min} > F_{gmu}$. If the frequency of GMU instructions is small, then this may not cause a problem. However, if the objective is made to reduce the overhead of goal management execution, R_e , below the value K , where $K < F_{gmu}$ this cannot be achieved if $t_{gmu}^h = 1$.

Therefore, to reduce the overhead of goal management below F_{gmu} , the effective execution time of a goal cache hit must be set to zero. That is,

$$t_{gmu}^{e,h} = 0 \quad (8.33)$$

The above condition of making the effective execution time of GMU instructions that result in a goal cache hit, equal to zero can be achieved by previously prefetching instructions from the continuing instruction stream. This issue was discussed in Chapter 7.

Expression (8.27) is now rewritten as follows:

$$R_e^{min} = \frac{\bar{W}}{(1 - F_{gmu})\bar{t}_{ru}} \leq K \quad (8.34)$$

which sets the following requirement for the average RU-GMU wait time, \bar{W} :

$$\bar{W} \leq K \times (1 - F_{gmu})\bar{t}_{ru} \quad (8.35)$$

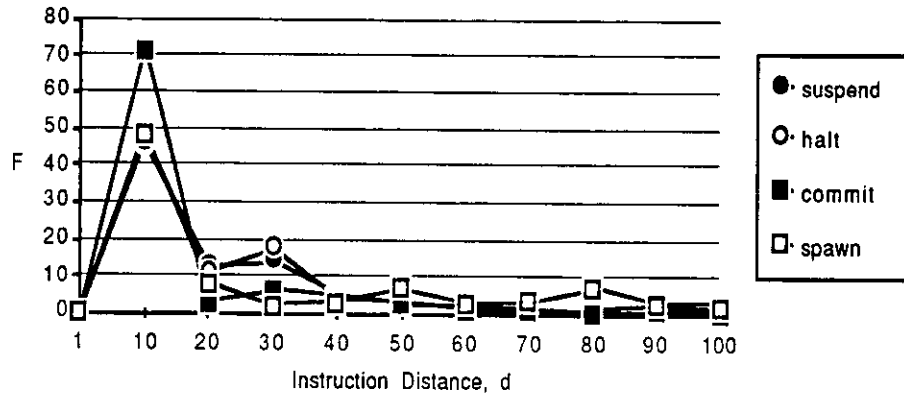


Figure 8.4: GMU Instruction Distance Distribution

To set the boundary condition for the average RU-GMU wait time, let us again consider the case where each RU instruction executes at an average rate of one instruction per cycle ($\bar{t}_{ru} = 1$). The condition set by expression (8.35) now becomes:

$$\bar{W} \leq K \times (1 - F_{gmu}) \quad (8.36)$$

To determine the range of architectural parameters required to obtain an average instruction wait time that satisfies condition (8.36) and to be able to evaluate the GMU utilization, we now consider the distribution of GMU instructions in the large FCP benchmarks described in Chapter 5.

8.6.1 Average RU-GMU Wait Time, \bar{W}

In Figure 8.4 we show the distribution of instruction distances between consecutive GMU instructions *halt*, *spawn*, *suspend* and *commit*. In all cases, 90% of GMU instructions are followed by another GMU instruction in less than 30 RU instructions.

Based on the distribution shown in Figure 8.4, and using expression (8.11), we compute the average GMU instruction execution wait time \bar{W} , shown in Figure 8.5, for 3 different cases of goal memory bandwidth and two cases for the goal cache size. The goal memory bandwidth considered is 2, 4 and 8 bytes/cycle. The two goal cache sizes are a minimal cache that consists of 4 windows: *active*, *spawn ready* and *free*, and a goal cache that is *large*. A large goal cache enables all halt and spawn instructions to always execute in the goal cache. We do not

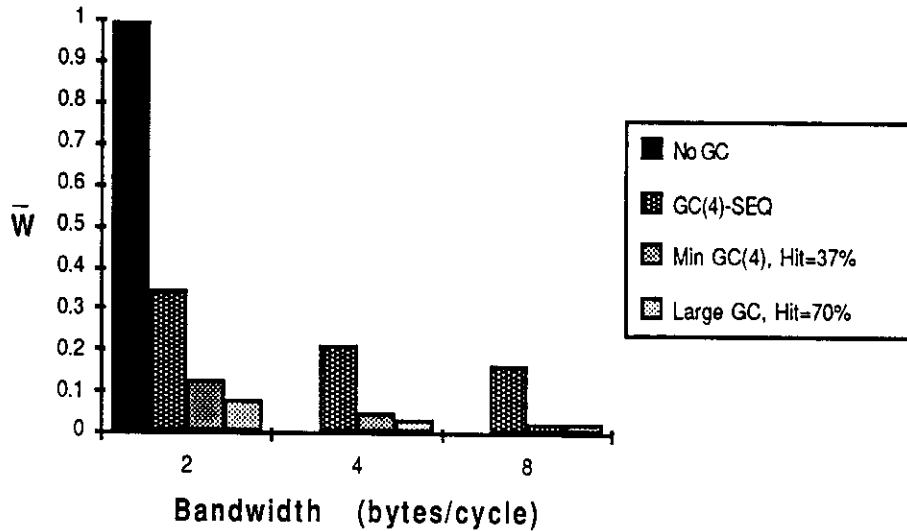


Figure 8.5: Average Wait Time, \bar{W}

correlate the actual goal cache size and the captured locality of halt and spawn, but just examine the two extreme cases.

Also shown in Figure 8.5 are the following two special cases. In the first column we show the average wait time when the goal management unit and the goal cache are not used. In this case the goal management operations are implemented *in software*, using the FCP processor instruction set. We label this average wait time as \bar{W}_1 . Since this average wait time does not depend on the goal memory bandwidth, we only show it in the first column, that is, for the goal memory bandwidth of 2 bytes/cycle. We will elaborate more on this issue later in this section.

In the second column, we show the average wait time when the goal management operations execute sequentially using the same execution times as if a minimal goal cache is used. We label this average wait time as \bar{W}_2 . The second and third column represent the average wait time for a minimal cache size \bar{W}_3 , and a large cache size \bar{W}_4 . The difference between the second and the third column in Figure 8.5 is only that the goal management operations are overlapped.

Goal Management Without Architectural Support

If the goal management operations are implemented in software using the FCP processor instruction set, in Figure 8.5 we show that the average wait time is almost 1. This implies that half the time of goal reduction is spent performing goal management. The following features affect the execution time of goal management operations when they are implemented without a goal management unit. We list only few of the most important features.

- If a goal cache is not used during goal reduction, all goal arguments are manipulated in memory. Even if processor registers are used during a goal reduction, a goal that is scheduled has to bring all the goal arguments into processor registers. This is avoided if the the goal cache is used.
- Spawning a goal requires the allocation of a goal record, which is performed by executing processor instructions. Using the goal management unit, this operation is efficiently performed by selecting the next free window in the goal cache.
- Without goal windows, modified goal arguments stored in processor registers must be written back into the goal record stored in memory, prior to goal suspension. This is avoided when the goal cache is used. The reason is that the goal cache is an extension of the goal queue in memory. A goal that is brought into the goal cache from the goal queue, is no longer represented in goal memory. This is not the case when the complete queue is stored in memory. The active goal, if it is first brought into processor registers and then modified, must be written back into the goal memory.

Another important feature accounts for the degree of goal management relative execution time in the case when goal management operations are implemented in software. The fact that the processor architecture has support for processor dereferencing and argument trailing, as well as tag manipulation, reduces the effective time that goal reduction is being performed, compared to an implementation on a general-purpose processor where this support is not available. This then results in the increased amount of relative goal management execution time. Moreover, the result would be even more in favor of goal management execution if an improved compiler is used that would reduce the effective time of clause-try execution.

Sequential Execution using Goal Cache

Using the goal cache and the goal management algorithms proposed in Chapter 7, and implementing all goal management operations in a non-overlapped mode of execution reduces the average wait time significantly. If we consider the ratio of the average wait time \overline{W}_1 and \overline{W}_2 for the case when the goal memory bandwidth is equal to 2 bytes/cycle:

$$S_1 = \frac{\overline{W}_1}{\overline{W}_2} = 2.96 \quad (8.37)$$

we see that more than a three fold reduction in the average wait time is achieved using the goal cache. The reduction in the average wait time further increases by increasing the goal memory bandwidth.

Overlapped Execution using Goal Cache

By overlapping goal management execution using the goal cache, further reductions of the average wait time is obtained. For the goal memory bandwidth of 2 bytes/cycle, the ratio:

$$S_2 = \frac{\overline{W}_2}{\overline{W}_3} = 2.75 \quad (8.38)$$

results in an improvement of almost four times, for the minimal cache configuration. Further reductions in the average wait time are obtained by increasing the goal cache size, but these changes are not significant. The overall reduction of the average wait time, from the software implementation to the overlapped execution using a goal cache is denoted as the product:

$$S = S_1 \times S_2 = 8.14 \quad (8.39)$$

Therefore, almost an order of magnitude of reduction in the average wait time is achieved. We now further discuss the behavior of the average wait time with respect to the selected goal cache algorithm and the goal cache size.

Increasing Goal Cache Size

Given the goal cache algorithm described in Chapter 7, the goal cache size does not influence the average execution time of the suspend instruction. Every goal that suspends is moved to the goal memory. Therefore its execution time

depends only on the goal memory bandwidth. The goal cache size influences the average execution time of halt and spawn. However, most of the halt and spawn instructions are overlapped even with a minimum goal cache configuration of 4 goal windows. The difference between the minimum goal cache and a large goal cache becomes even less significant as the goal memory bandwidth increases, since the execution times of halt and spawn are further reduced. The wait time is almost completely the result of how goal suspension is implemented. For the goal memory bandwidth of 2 bytes/cycle, the reduction in the average wait time obtained by increasing the goal cache size is represented as S_3 , where:

$$S_3 = \frac{\overline{W}_3}{\overline{W}_4} = 1.8 \quad (8.40)$$

Another interesting observation is that the increase in goal memory bandwidth does not reduce the wait time as much as one may expect, assuming that the average execution time of goal suspension is memory bound. That is, one would expect the wait time to be zero. However, increasing the goal memory bandwidth reduces the time it takes to transfer a goal from the goal cache to the goal memory, which is enough to reduce the effective execution time of halt and spawn. However, the complexity of goal suspension requires the allocation of suspension notes that are then linked into suspension lists. This implies an inherent data dependency that does not depend only on the memory bandwidth. Increasing the goal memory bandwidth reduces the goal suspension time as long as it affects the transfer time from goal cache to goal memory. Further increases in goal memory bandwidth do not have any effect on goal suspension execution time.

From the results shown in Figure 8.5 a minimum goal cache of size 4 goal windows, together with a goal memory bandwidth of 4 bytes/cycle results in a system with an average wait time of less than 4%.

Assuming that IU prefetches the next instruction using the next window pointer (NWP) set by GMU ($\bar{t}_{gmu}^{e,h} = 0$), and that RU executes a RISC-type instruction set ($\bar{t}_{ru} = 1$), the average instruction execution time $\bar{\tau}$ is represented as:

$$\bar{\tau} = 1 - F_{gmu} + \overline{W} \quad (8.41)$$

With the frequency of goal management operations being $F_{gmu} = 3.6\%$, the average instruction execution time is $\bar{\tau} = 1.014$. Therefore, if RU executes one instruction per processor cycle, and all goal management operations are overlapped

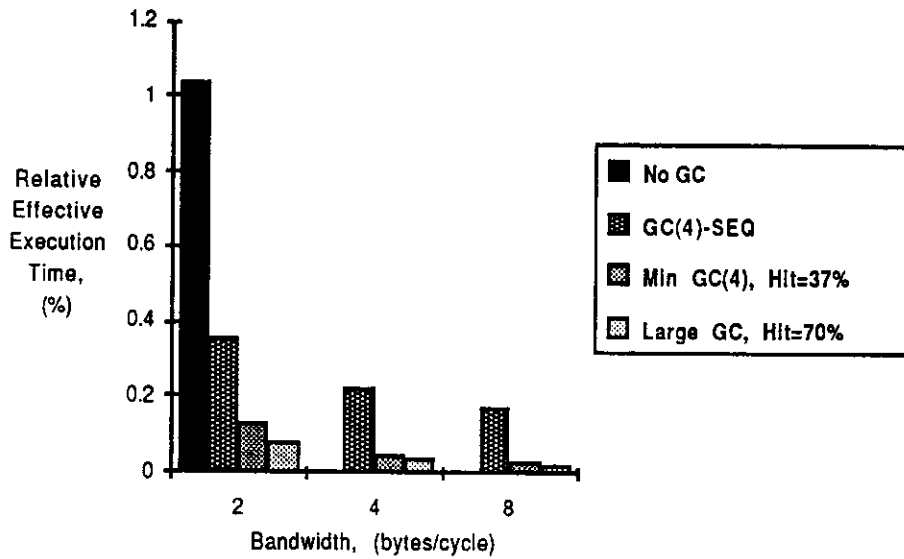


Figure 8.6: Relative Effective GMU Execution Time, R_e

using a minimal goal cache with a goal memory bandwidth of 4 bytes/cycle, the effective execution time of goal management operations is close to one per processor cycle. The degradation relative to the single-cycle execution rate is less than 2%.

8.6.2 Relative Effective Execution Time, R_e

As expected, the Relative Effective Execution time, shown in Figure 8.6 is affected by both the goal cache size and the goal memory bandwidth. As the bandwidth increases, the effect of the goal cache is reduced. For a goal memory bandwidth of 4 bytes/cycle and a minimal goal cache configuration, the relative execution time is 5%. Further increases in bandwidth result in a relative execution time values that are close to 2%.

When the goal management operations are implemented in software, the relative execution time of goal management execution is over 1, meaning that over 50% of the execution time is spent performing goal management operations. This is reduced to 20% when the goal memory bandwidth is 4 bytes/cycle and sequential goal cache is used. By overlapping goal management operations in the goal cache, the relative execution time is reduced to less than 5%, for the same goal memory bandwidth value.

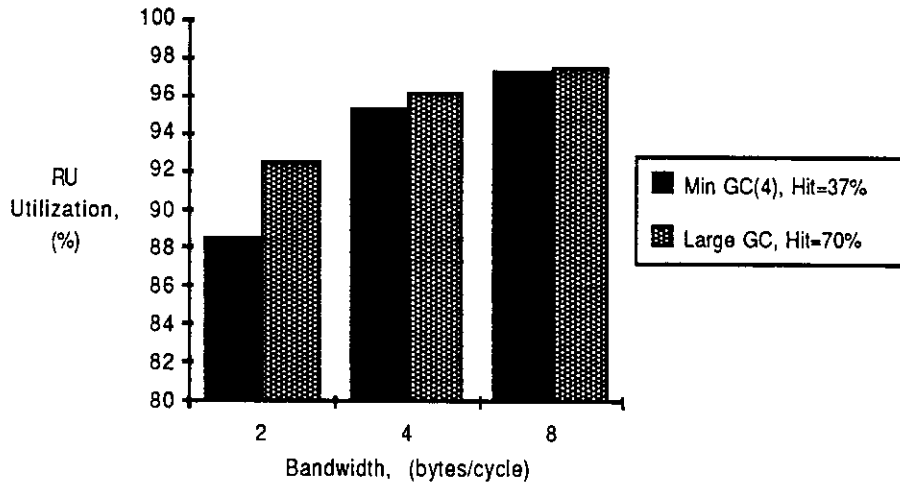


Figure 8.7: RU Utilization, U_{ru}

8.6.3 RU and GMU Utilization, U_{ru} , U_{gmu}

In Figure 8.7 we show the utilization of RU which is always over 88%. As the goal cache size or the goal memory bandwidth increase, RU utilization increases since the average execution times of the goal management operations are reduced, and thus the RU wait time is reduced. This is not the case for the GMU utilization shown in Figure 8.8. As the average execution time of GMU instructions is reduced, so is the GMU utilization. This is because the reduction of goal instruction execution time is much more significant than the reduction of the resulting wait time and thus the total program execution time.

For example, for the minimum goal cache size and a goal memory bandwidth of 4 bytes/cycle, the GMU utilization is $U_{gmu} = 20\%$. This implies an imbalance of goal management and goal reduction leading to an underutilized GMU. A better utilization of GMU could be achieved at the expense of program execution time by making GMU slower. However, this would result in an increase in the RU-GMU wait time and thus performance degradation.

The meaningful direction to improve GMU utilization is to allow GMU to perform additional useful computations while it is idle. These operations should be of lower priority, so that when the requests for goal management operations arrive, they are not delayed. For example, GMU may implement more complex

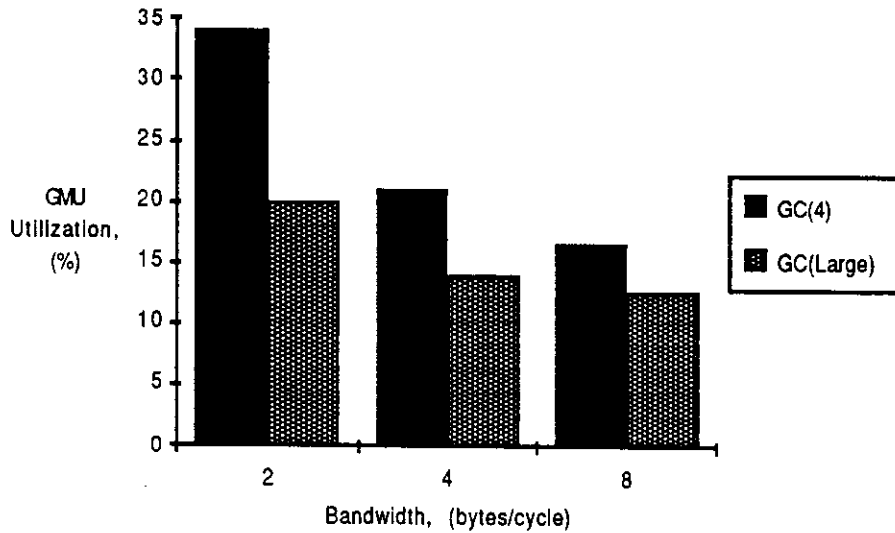


Figure 8.8: GMU Utilization, U_{gmu}

scheduling algorithms that involve the management of *goal priorities*. For example, if one distinguishes between user goals, system goals and kernel goals, GMU may compute in advance the scheduling priorities. Another example may be the support of real-time systems where time-critical goals are scheduled first.

A more heuristic approach could be to dynamically keep count of statistics used for scheduling. While GMU is idle, it could access the goal queue and reorder goals according to the statistics. One possibility is to keep the count of the number of suspensions a goal makes during program execution. Goals that communicate with each other frequently, could be *bubbled-up* closer to the front of the queue from where they are scheduled for execution.

8.6.4 Chapter Summary

We have evaluated the overlapped execution of goal reduction and goal management in the proposed FCP processor. The evaluation is based on specific performance measurements and analytic models that define the relation between the performance measures and other system parameters. The main result is that the relative execution time of goal management can be reduced to less than 5% compared to the sequential execution with no goal management support. This represents close to an order of magnitude reduction in the relative execution time. This is achieved by first providing the special-purpose goal cache and then defin-

ing the overlapped execution model. It is assumed that the first instruction in the next goal is prefetched by the Instruction Unit, and that all RU instructions execute in a single processor cycle. The goal memory bandwidth requirement was set at 4 bytes/cycle.

Using the goal cache algorithm specified in Chapter 6 and 7, the goal cache size does not significantly affect the goal management relative execution time and average wait time. A minimum cache size of four goal windows achieves adequate performance, given the goal memory bandwidth of 4 bytes/cycle.

The RU utilization is evaluated to be always above 88%, whereas the GMU utilization is close to 20%. We have discussed ways of increasing the GMU utilization by performing useful operations during its idle cycles.

CHAPTER 9

FCP Processor Performance Evaluation

We now evaluate the FCP processor architecture described in Section 6 using the analytic performance evaluation model defined in Chapter 5. The same parameters that represent the System's Development Workload are used here. However, the analysis in this chapter differs from that presented in Chapter 5 in the following way.

- The analysis of potential implementation bottlenecks described in Chapter 5 was performed at the abstract machine implementation level, since it was important to have results that were independent of the host physical machine and abstract machine emulation language. All implementation dependent parameters in the model were treated as variables, with a range of implementation dependent values.

The analysis described in this chapter, is performed after the FCP processor architecture is specified. Therefore, low-level analysis is justified, since we are now interested in optimizing a specific architecture, that is, the proposed FCP processor. Rather than looking at a range of values for the implementation dependent parameters, we consider specific values.

We present the performance evaluation of the FCP processor in the following three steps. First, we discuss those aspects of the FCP processor architecture that determine the values of the implementation dependent parameters defined in Chapter 5. In the second section, we consider how each functional unit separately contributes to system performance, followed by their combined effect on performance. In the third section we analyze how the FCP processor architecture performs for a variety of workloads.

9.1 Implementation Dependent Parameters

In Chapter 8, we analyzed in detail the performance of overlapped goal management in the FCP processor. We also described the values for the implementa-

tion dependent goal management parameters that correspond to the halt, spawn, suspend and commit instructions: \bar{t}_h , \bar{t}_{sp} , \bar{t}_{susp} and \bar{t}_{com} .

We now discuss the following goal reduction implementation dependent parameters that are defined in Chapter 5:

- \bar{t}_g : Average argument matching execution time.
- \bar{t}_p : Average argument creation execution time.
- \bar{t}_d : Average argument dereferencing execution time per unit length.
- \bar{t}_t : Average variable trailing execution time.

By executing a RISC instruction set with an effective throughput of one instruction per cycle, the average execution time of the clause-try operation per argument as well as the average execution time of creating a new goal argument, is equal to the average number of executed instructions per operation. The instruction counts are specified in Chapter 8.

The dereference operation per unit length consists of fetching a word from memory, isolating the tag value and checking whether it is a reference or not. Since RU executes together with the tightly-coupled TU, tag manipulation operations, such as tag decoding of a word loaded from memory, are performed concurrently. In this case, the execution time of the dereference operation per unit length is equal to 1 RISC load instruction cycle. That is, we consider $\bar{t}_d = 1$.

In the FCP processor, the data trailing operations are part of the Data Cache policy. Let us first consider the execution time of a trail function without the Data Cache. In this case, trailing is performed in the Data Memory using a separate Trail Stack, TS. Trailing a single variable consists of 2 instructions to store the trailed address and value. We assume the values were previously loaded into processor registers during program execution. That is, it is not necessary to first load the address and value into registers. In case of failure or suspension, undoing the trail takes an additional 2 load instructions, to bring the address and value from TS into registers, followed by 1 store instruction to restore the old memory word value. In case of clause-commit, TS is reset, and then there is no need for additional instructions.

Therefore, trailing during a clause-try that succeeds requires 2 memory instructions, and during a clause-try that fails or suspends requires 5 memory

instructions, when there is no architectural support for data trailing. From measurements in Chapter 5, we showed that most of the trailing occurs during those clause-tries that succeed. For the System’s Development Workload, the average number of trailed values during a clause that commits is equal to 5. The amount of trailing performed during those clause-tries that fail or suspend is an order of magnitude lower.

With the use of the Data Cache, trailing is performed in a transparent way, as part of the Data Cache policy. No penalty is incurred as long as the number of trailed elements in the cache does not result in overflow. If it does, the clause-try is restarted with the trailing policy performed in the Data Memory.

9.2 Performance Improvements due to Functional Units

We now evaluate the special-purpose FCP processor, by considering the contribution of each functional unit to system performance. Let T denote the program execution time in a system that has no special-purpose support for function f , and let T_f denote the execution time of the same function. The upper bound on speedup due to a special-purpose functional unit for function f is given by the following expression:

$$S_f = \frac{T}{T'_f} \tag{9.1}$$

where T'_f denotes the execution time when an *ideal* functional unit is available. That is, an ideal functional unit reduces the program execution time by the execution time of function f , T_f , as follows:

$$T'_f = T - T_f \tag{9.2}$$

According to the definition of the Relative Execution Time of function f , R_e^f , given in the previous chapter, the speedup is represented as:

$$S_f = 1 + R_e^f \tag{9.3}$$

In the following analysis, we first consider the execution of only RU and TU, without any support for goal management or data trailing. However, the system has support for dereferencing in the form of a special-purpose instruction. We then consider the contribution of units for goal management and data trailing.

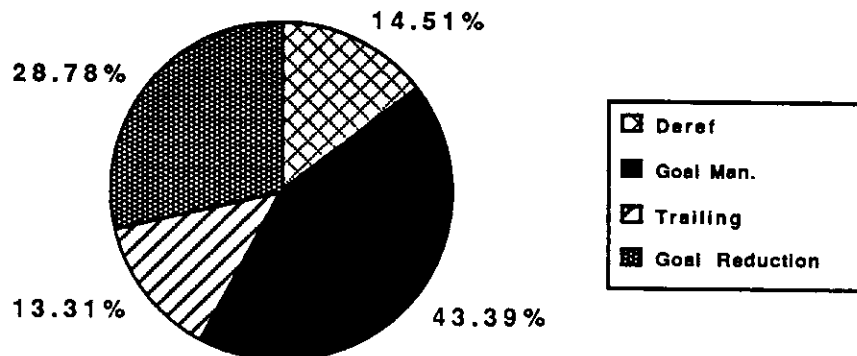


Figure 9.1: Relative Execution Times of Goal Reduction Functions

By implementing each goal management operation using FCP RISC instructions and using the workload parameters of Section 5, we evaluate the relative execution times of goal management, argument dereferencing, variable trailing and the remaining execution time of goal reduction, as shown in Figure 9.1. The goal management is implemented in software, that is, without the support of the goal management unit. We see that almost half the execution time is spent performing goal management. Argument dereferencing and variable trailing together contribute as much as the remaining part of goal reduction. From this diagram, we reconfirm that goal management does represent the system-bottleneck.

Since Figure 9.1 corresponds to a processor that does not have support for goal management and data trailing, but does have support for dereferencing and tag manipulation, the execution environment subsumes the architecture of the *Carmel* processor described in [Hars88]. Therefore, for the System's Development Workload, and an execution environment that consists of RU and TU, the main system bottleneck is goal management. The maximum speedup due to special-purpose units for goal management and data trailing are evaluated for the specific workload as follows:

$$S_{gmu} \approx 2, S_{trail} \approx 1.2 \quad (9.4)$$

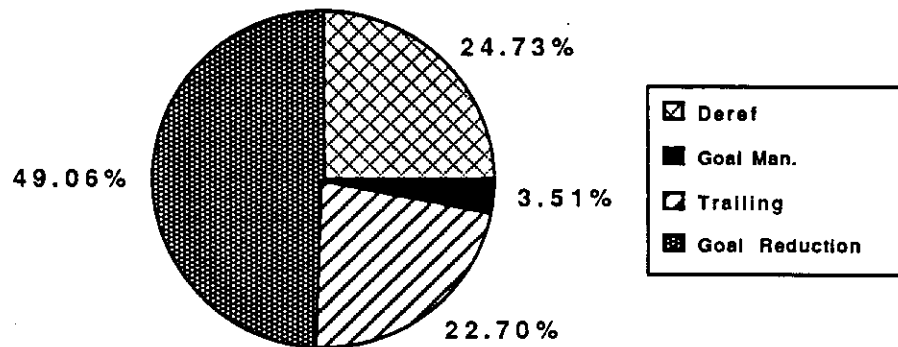


Figure 9.2: Relative Execution Times with Support for Goal Management

9.2.1 Support for Goal Management

In Chapter 8, we showed how the relative execution time of goal management can be reduced to approximately 3-4% of program execution using a special-purpose goal cache and an overlapped goal management policy. The required goal memory bandwidth was set at 4bytes/cycle. Compared to an FCP processor architecture that does not have support for goal management, the relative execution time is reduced an order of magnitude.

In Figure 9.2 we show the relative execution times when the architectural support for goal management operations are added. The system bottleneck shifts from goal management to the remaining goal reduction operations. Also, the relative execution times of argument dereferencing and variable trailing become more significant, even though they are not the main bottleneck. Therefore, the new bottleneck of system performance is the goal reduction time which consists of manipulating data objects during a clause-try and during the creation of goal arguments.

9.2.2 Support for Data Trailing

From Figure 9.2 we also see that the relative execution time of data trailing is slightly less than dereferencing. However, the dereferencing operations are not an overhead, but a part of the data manipulation operations. The dereferencing

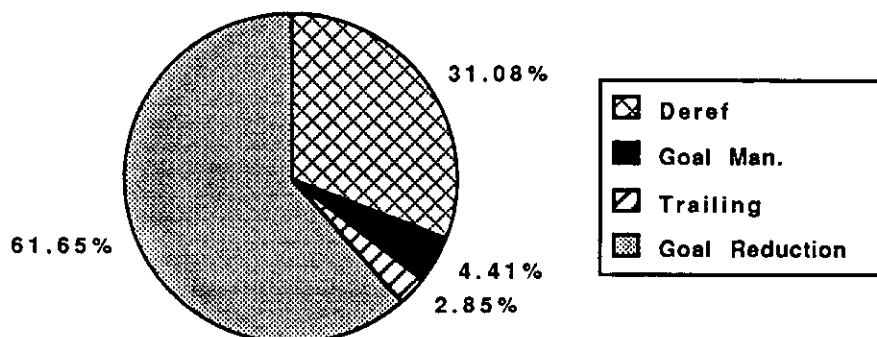


Figure 9.3: Relative Execution Times with GMU and Data Trail Support

operations are already optimized and are equivalent to single memory load operations. By adding the special-purpose Data Cache with the *Delayed Binding* policy, the program execution time can be improved by at most 22%. That is, the achievable speedup is:

$$S_{trail} = 1.28 \quad (9.5)$$

Combined together, the maximum speedup due to goal management and data trailing is given by:

$$S = S_{gmu} \times S_{trail} = 2.56 \quad (9.6)$$

relative to a processor that has no support for goal management and data trailing.

In Figure 9.3 we show the distribution of the relative execution times when there is support for goal management (GMU) and data trailing (Data Cache). The overhead of data trailing is computed as follows. For all cases where the number of trailed elements is less than 10, we assume that the Data Cache performs shallow backtracking using the *Delayed Binding* cache policy. For all cases where there are more than 10 trailed elements, it is assumed that the trailing is performed in the Data Memory. If the trailing occurred during a clause-commit, the penalty is 2 instructions per trailed element, otherwise it is 5.

After providing architectural support for both goal management and data trailing, what remains is the relative execution time due to goal reduction. We now consider ways of further improving the FCP processor system performance.

9.3 Performance versus Goal Management Complexity and Goal Reduction Granularity

The fact that the Goal Management Unit can double the performance of the RU+TU execution environment, that is, an execution environment that does not have support for goal management, is not unreasonable. On the contrary, the specific workload that we consider in this thesis is not particularly favorable to the special-purpose GMU. We now discuss in more detail the following issues, and their effect on system performance:

- **Granularity** of Goal Reduction
- **Granularity** of Goal Management
- **Complexity** of Goal Management

Granularity of Goal Reduction

By granularity of goal reduction we mean the average execution time of goal reduction during program execution. We label this execution time as T_r . For simplicity, let us consider an execution environment in which each program execution (interpretation) step executes in a single cycle, that is, a RISC-type instruction set. In this case, the granularity or average goal reduction execution time, is equal to the number of executed RISC instructions, N_r .

Compared to previously reported measurements, the programs used in the System's Development Workload exhibit a higher average granularity of goal reduction. The reason is that our programs are in most cases list oriented operations that do not explicitly model inter-goal communication and synchronization. Therefore, the granularity is representative of the specific application area for system development.

In Table 9.1, we show the distribution of the number of RISC instructions executed (IC) per goal reduction (GR). The average value is equal to 256 instructions per goal reduction. Note that the number of RU instructions do not include the GMU instruction count. The higher granularity of goal reduction resulted in a relatively low frequency of goal management operations. In Chapter 5, we show that the average value is $F_{gmu} \approx 4\%$.

FCP Benchmark Programs							
	Compiler	Sim1	Sim2	Debug	Solver	Distr.	Logix
GR	7075380	3009280	7722376	1593286	409075	259283	1481900
$\frac{IC}{GR}$	25	43	20	485	728	416	78

Table 9.1: Goal Reduction Granularity

However, one should also note that there is a significant difference in the granularity between the various applications programs. For example, for the *Compiler*, the granularity is 25 instructions, and for the *Solver* it is 728 instructions. This can be explained as follows. The programs that have a higher granularity in the System's Development Workload are applications for FCP program interpretation, that is *meta-interpreters*. For example, the program *Debugger* is used to debug the execution of the FCP processor simulator written in FCP. In the simulator, a goal reduction consists of modifying the processor state for each fetched and executed instruction. This results in a low average granularity. However, the Debugger treats the simulator as data, and symbolically interprets its execution. Thus the high-granularity.

As part of the System's Development Workload, we did not consider the two types of applications separately, since we defined the workload as consisting of a mix of both types of applications. That is, the user is not just running simulations or meta-interpreters, but doing both.

Granularity of Goal Management

The selected FCP programs in the System's Development Workload exhibit a higher goal suspension and activation rate than previously reported in [Tick88]. In Chapter 5, we evaluated the Average Goal Management activity (AGM) to be 1.3 goal management operations per goal reduction. This includes goal creation, goal termination, goal suspension and activation. The most time consuming goal management operation is goal suspension. We showed that on the average, goals in the selected programs suspend on 2.6 variables. This was also higher than originally anticipated.

The *granularity* of goal management, labeled as T_{gm} , depends on several program and implementation dependent parameters. The simplest way to measure the granularity of goal management is in a way analogous to measuring goal granularity. That is, by evaluating the effective number of RISC-type instructions,

N_{gm} , required to perform the average number of goal management operations per goal reduction. Program parameters such as the average number of suspension variables affect the execution time of goal management according to the performance models defined in Chapter 5. Therefore, we say that a program exhibits a *higher granularity* of goal management (compared to another program), if the number of interpretation steps using a RISC-type instruction set is higher.

Complexity of Goal Management

We define the complexity of goal management \mathcal{C} as the ratio of goal management execution time T_{gm} , and goal reduction execution time, T_r . That is, the ratio of goal management and goal reduction granularities:

$$\mathcal{C} = \frac{T_{gm}}{T_r} \quad (9.7)$$

Thus, when we refer to applications with higher complexity of goal management, it means that they have a higher execution time of goal management relative to goal reduction.

For example, in [Tay189], a set of five FCP programming stereotypes were selected for benchmarking. These programs have a low granularity of goal reduction and goal management, but a high complexity of goal management. The applications explicitly model inter-goal communication protocols and thus perform frequent goal management operations, whereas the goal reduction phase is simple. However, the ratio of goal management execution time versus goal reduction execution time can be very high.

In Figure 9.4, we symbolically represent the performance of a system workload in the space of goal management and goal reduction *granularities*. Both measures of granularity (execution time), are implementation dependent. That is, they depend on whether the execution environment (interpretation mechanism) is a RISC, CISC, VLIW architecture etc. We consider for both features that the execution environment is a RISC-type instruction set where instructions execute in a single cycle. For a given processor cycle time, the number of RISC-type instructions executed is directly proportional to the execution time. Represented in Figure 9.4 are the following concepts, for system workloads that execute the same number of goals, that is, $N_g = const.$

1. An application domain that exhibits an increased goal management granularity for the same goal reduction granularity results in performance degra-

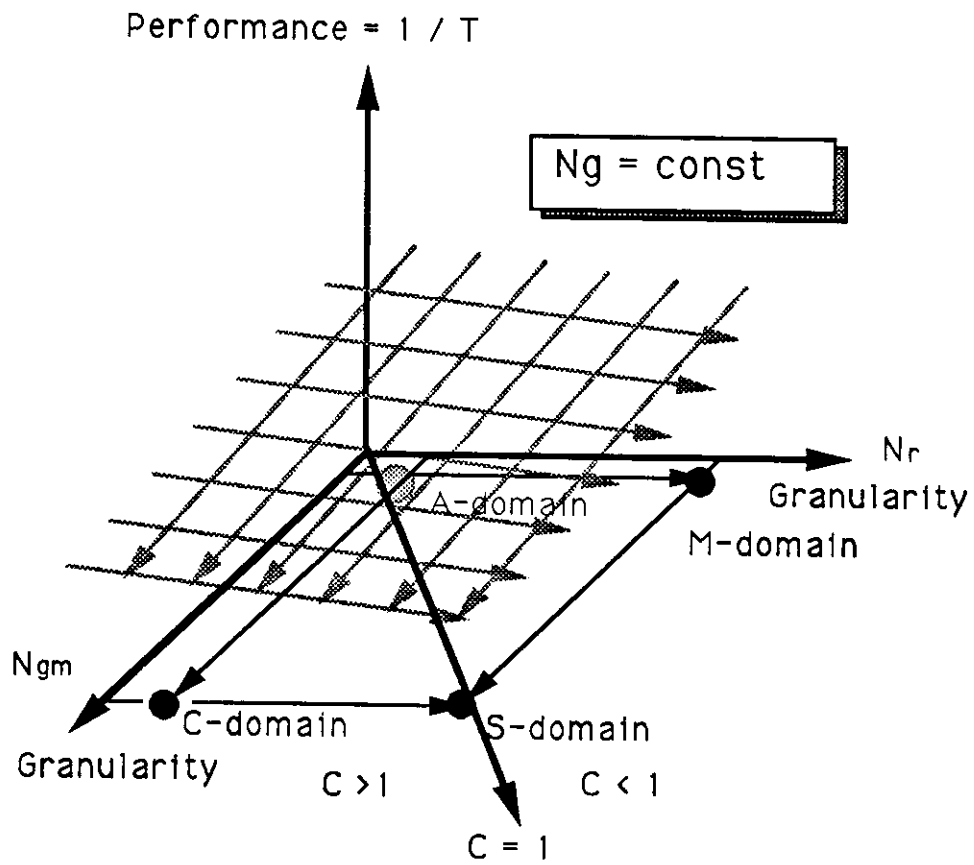


Figure 9.4: Performance of Different Application Domains

dation. This is shown in Figure 9.4 by *slanting* the performance plane towards higher granularity of goal management.

2. An application domain that exhibits lower granularity of goal reduction for the same degree of goal management activity results in an increase in system performance. This is indicated by showing the performance plane *slanted* towards higher granularity of goal reduction.

The first issue corresponds to the following example. Consider two applications with the same granularity of goal reduction, but in the first program each goal suspension suspends on one variable, whereas in the second application each suspension suspends on two variables. It is clear that the granularity of goal management is greater in the second program, and thus the system performance of the first application is better than the second application.

In the second case, consider two applications that have the same goal management execution times, but the first executes less instructions per goal reduction than the second. It is again clear that the first application exhibits better system performance.

Both goal reduction and goal management granularities are values greater than zero. For each goal that is reduced, it at least has to be spawned and terminated, which we consider as part of goal management.

Application Domains

Depending on the goal management and goal reduction granularities, we partition the space shown in Figure 9.4 into the following four regions:

- M-domain: High goal reduction and low goal management granularity.
- A-domain: Low goal reduction and low goal management granularity.
- C-domain: Low goal reduction and high goal management granularity.
- S-domain: High goal reduction and goal management granularity.

The M-domain denotes the region of low goal management and high goal reduction granularity (more than 200 instructions). This domain is observed

to be characteristic of applications such as *Meta-Interpreters*, which perform symbolic interpretation of programs as data.

The application domain with both low goal management and goal reduction granularity is labeled as the A-domain. A typical example of a program in this domain is the deterministic list *Append* program, that has often been used for comparative benchmarking. This application performs very little goal management (no goal suspension), thus exhibiting low goal management granularity. The granularity of a goal reduction in the A-domain is approximately 20 RISC operations.

In contrast, the *System's Development Workload*, has an average goal reduction granularity that is similar to the M-domain, but a goal management granularity that is significantly higher than the A-domain. We labeled this workload domain as the S-domain.

The C-domain denotes the region of applications with high goal management granularity and low goal reduction granularity. Applications characteristic for this domain are *Communication* protocols that explicitly model the inter-goal communication and synchronization.

Also shown in Figure 9.4 is the line that delimits the domains of higher complexity ($C > 1$) from lower complexity ($C < 1$). The complexity line ($C = 1$) marks the domain where the execution time of goal management is equal to the execution time of goal reduction, that is, $T_{gm} = T_r$.

In the following analysis, we consider the effect of overlapped goal management execution, on program execution time, in the space of alternative goal management and goal reduction granularities, as well as complexities.

9.4 Overlapped Goal Management versus Granularity and Complexity

In Figure 9.5 we label the average program execution time of the *System's Development Workload* as T . It consists of the goal reduction execution time, T_r , and goal management execution time T_{gm} . Without goal management support, we showed in Figure 9.1 that the goal reduction time is almost equal to the goal management time, that is, $T_r \approx T_{gm}$. In other words, the complexity of goal management for the S-domain and the *System's Development Workload* is $C \approx 1$.

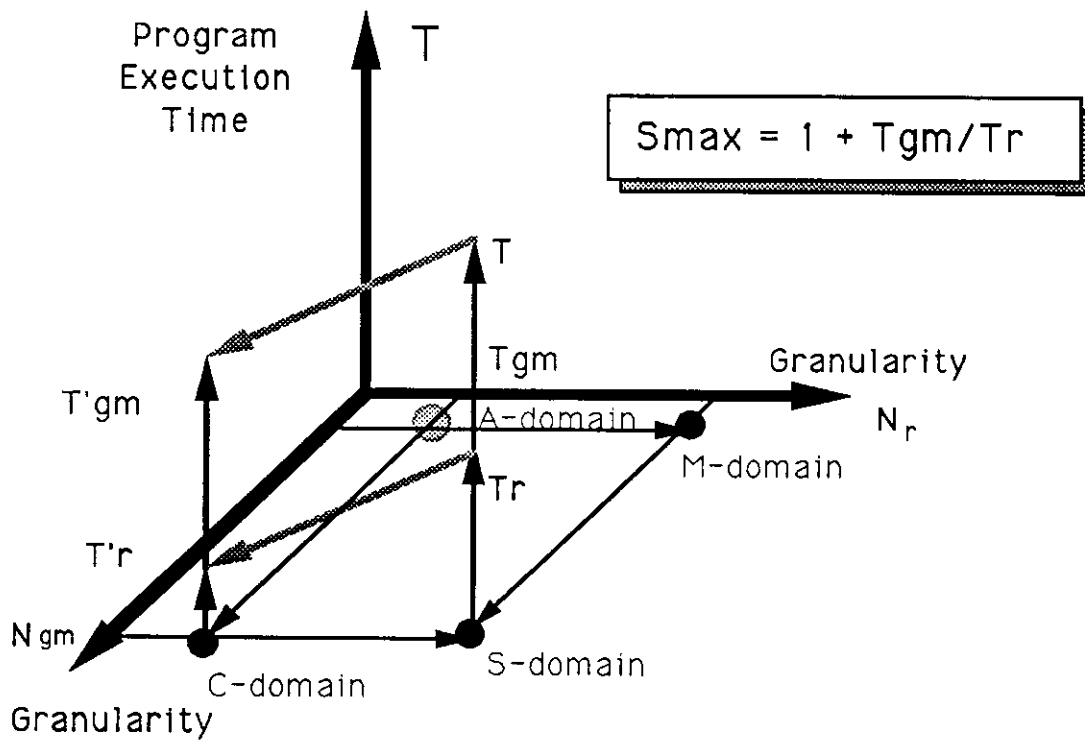


Figure 9.5: Maximum Speedup for Different Application Domains

In this case, the goal management operations were considered a bottleneck, which motivated special-purpose support using GMU. The maximum possible speedup due to overlapped execution of goal management, S^{max} , is achieved when the goal management operations are completely overlapped using GMU, resulting in zero wait time by RU. That is:

$$S^{max} = 1 + \frac{T_{gm}}{T_r} = 1 + C \quad (9.8)$$

Realistically, however, a delay of W results from the overlapped execution using GMU. In this case, the obtained speedup is represented as:

$$S^r = \frac{T_{gm} + T_r}{T_r + W} = \frac{1 + C}{1 + \frac{W}{T_r}} \quad (9.9)$$

For the System's Development Workload where the complexity is $C = 1$, the maximum speedup, denoted as S_S^{max} , is $S_S^{max} = 2$. In Chapter 8, we showed that the wait time due to overlapped execution can be made small, less than 5% of T_r . Therefore, S_S^r is close to the maximum possible, since the operations are almost completely overlapped resulting in a low value for the overlapped wait time, W . Thus, $S_S^r \approx S_S^{max} \approx 2$.

In the C-domain (high granularity of goal management and low granularity of goal reduction), we label the goal reduction and goal management times as T_r' and T_{gm}' respectively. Since, for this application domain, it is true that $T_r' < T_r$, for the corresponding goal management complexities it is true that $C_C > C_S$. It thus results that the maximum possible speedup due to overlapped goal management using GMU in the C-domain, S_C^{max} , is:

$$S_C^{max} > S_S^{max} = 2 \quad (9.10)$$

Programs that are characteristic of this domain are applications that explicitly model inter-goal communication protocols. These applications spawn numerous goals that often suspend and activate without performing many goal reduction operations. In these cases, most of the execution time is spent performing goal management which consists of inter-goal communication and synchronization. From this point of view, the goals in this application domain are very tightly coupled. For this reason, several applications in [Tayl89] performed more efficiently on a single processor than the distributed implementation of general-purpose processors connected in a 16-node Hypercube machine.

We now answer the following two questions regarding the performance of overlapped goal management using GMU:

1. What is the effect of a faster RU on system performance?
2. What is the effect of scaling both goal management and goal reduction granularity?

9.4.1 Increasing RU Speed of Execution

We now consider the effect of increasing the speed of execution of goal reduction, for the same ratio of the number of goal management instructions versus number of goal reduction instructions. In other words, within the RU and GMU execution environment, we assume that RU execution time is scaled by a factor labeled s , where $0 < s \leq 1$.

A faster RU has the following two effects on execution times. First, RU executes goal reduction instructions faster, as well as goal management instructions in a system where GMU is not available. In other words, the software implementation of goal management operations is also faster when emulated by a faster RU. Second, a faster RU effects the distribution of goal management instructions which directly affects the RU wait time when GMU is available.

In Figure 9.6 we show the following three functions: the scaled goal reduction time $T_r s$, the RU-GMU Wait time $W(s)$, and the program execution time when RU and GMU execute concurrently, T . The third function is the sum of the previous two. That is,

$$T = T_r s + W(s) \quad (9.11)$$

As the speed of RU increases, the overall program execution time is reduced. However, the faster execution of RU instructions results in an increase in the RU-GMU wait time. For example, if the instruction distance between two consecutive goal management instructions was originally 20, now this distance is also scaled down and will result in more frequent and longer delays.

From Figure 9.6 we see that for a scale factor of 0.2, the wait time is more than half the goal reduction time, thus becoming again a bottleneck. Further increasing the RU speed results in almost completely wait time, and so the decrease in goal reduction time does not further affect system performance.

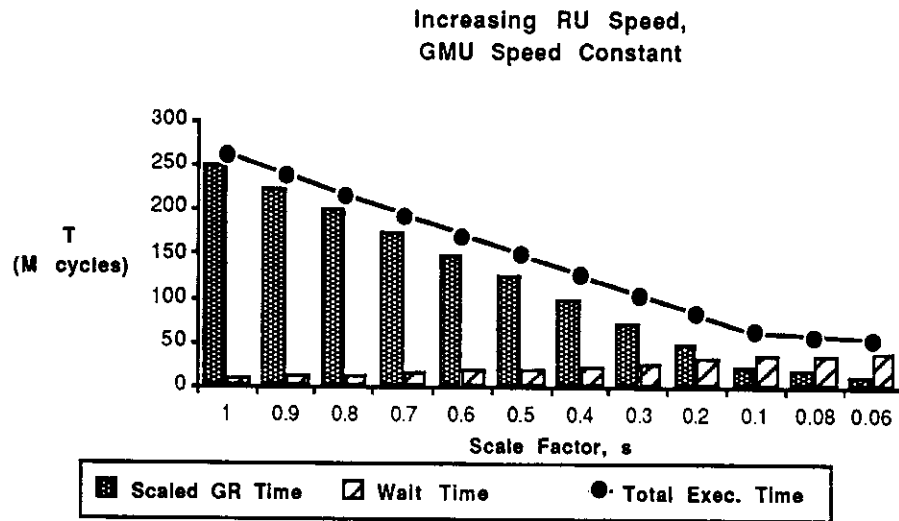


Figure 9.6: The Effect of a Scaled RU on Program Execution Time

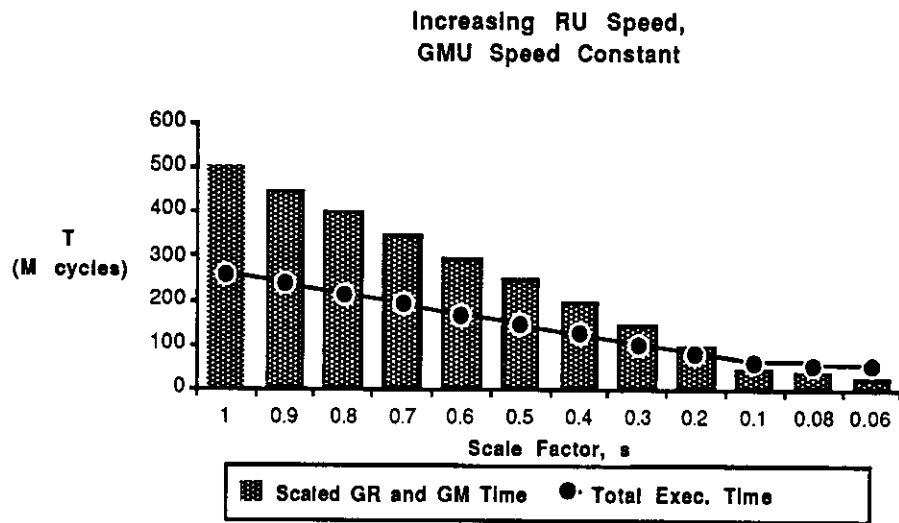


Figure 9.7: Scaled RU Execution Vs. Overlapped GMU

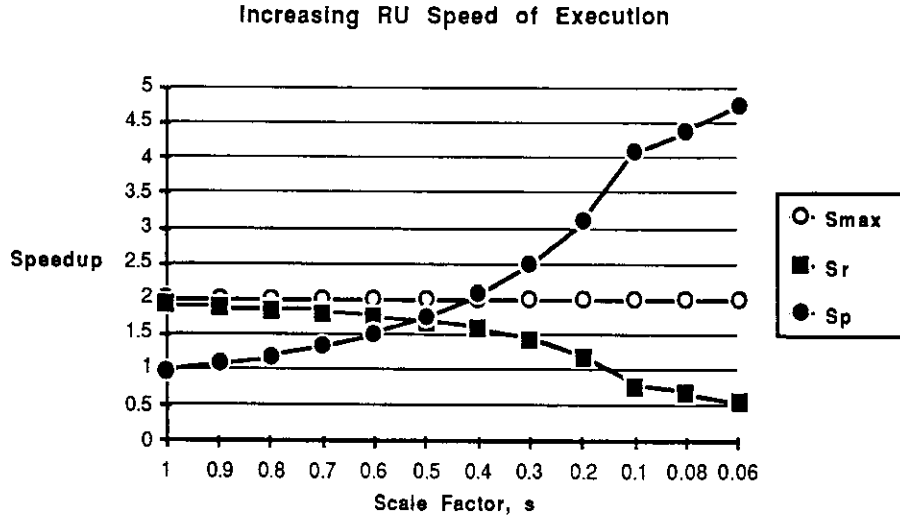


Figure 9.8: Speedup Versus Increased RU Speed of Execution

It is also interesting to consider how the increase in RU speed affects the execution of goal management operations when GMU is not used. That is, in Figure 9.7 we show the overlapped execution time as well as the software implementation time, T_{no_gmu} , when the scaled RU is used:

$$T_{no_gmu} = (T_{gm} + T_r)s \quad (9.12)$$

For the scale factor of 1, we see the same situation as described earlier, that is, the *software* implementation of goal management operations results in system degradation by a factor of 2. If RU performance increases by the scale factor, the overlapped execution time using GMU becomes closer to the execution time when GMU is not used. Moreover, when the RU speed is increased by a factor of 10, the overlapped execution is slower than the non-overlapped case with a faster RU. This is because we assume, in the overlapped case, that the speed of GMU does not change. Presumably, if one can increase the speed of RU execution one could do the same with GMU. However, with RU executing 10 times faster than GMU, RU implements goal management operations more efficiently than the overlapped execution using GMU.

The maximum speedup possible by completely overlapping goal management operations, relative to a system without GMU, is defined as follows:

$$S^{max} = \frac{(T_r + T_{gm})s}{T_r s} = 1 + C \quad (9.13)$$

Realistically, however, a wait time of $W(s)$ is incurred, and the speedup is expressed as:

$$S^r = \frac{(T_r + T_{gm})s}{T_r s + W(s)} = \frac{1 + C}{1 + \frac{W(s)}{T_r s}} \quad (9.14)$$

The two speedup functions are depicted in Figure 9.8. The maximum speedup is constant, since the complexity is constant. Moreover, since, for the System's Development Workload the complexity is close to 1, the maximum speedup is always equal to 2. By increasing the speed of RU execution and maintaining the same speed of GMU execution, the speedup decreases, and at some point results in system degradation. This is shown in Figure 9.8 where $S_r < 1$.

The above described behavior of S_r is intuitive. That is, as the speed of RU increases relative to a GMU whose speed is not changed, the benefit of overlapped execution using GMU is reduced. However, a faster RU does initially improve system performance, but, as the speed further increases, it is eventually degraded by the effect of growing wait time. To see how much the performance is improved by using a faster RU, in Figure 9.8 we also show a third function, S_p that denotes the relative execution time of a system with a scaled RU and without. That is,

$$S_p = \frac{T_r + W}{T_r s + W(s)} \quad (9.15)$$

We see that the speedup due to the increased speed of RU levels off at approximately 5.

From this analysis we conclude the following. Without GMU, we showed that goal management operations represent a serious implementation bottleneck. This bottleneck is eliminated by introducing the overlapped execution of goal management operations in GMU. Increasing the speed of RU execution results in performance improvements. If the speed of RU is increased by a factor of 10, a performance improvement of 5 is sustained. Further increases do not result in performance improvement, and goal management again becomes the system bottleneck.

9.4.2 Scaling Granularity

In the previous analysis, we considered scaling RU speed of execution without affecting granularity of goal management or goal reduction. In other words, the faster execution of RU also affected the software implementation of goal management operations, as well as goal reduction. We now consider increasing

the complexity of goal management by scaling down the execution time of goal reduction, and scaling up the execution time of goal management. In Figure 9.4, the scaling of goal management complexity represents moving to different application domains that are to the left of the line denoted by $C = 1$. Starting from the S-domain, the scaling moves to applications characteristic of the C-domain.

Using the same notation as before, let T_r denote the goal reduction time, T_{gm} the goal management time and g the granularity scale factor. If we first consider scaling down the goal reduction time, the program execution time T is expressed as follows:

$$T = T_r g + T_{gm} \quad (9.16)$$

The maximum speedup due to overlapped execution using GMU is then expressed as:

$$S^{max} = \frac{T_r g + T_{gm}}{T_r g} = 1 + \frac{C}{g} \quad (9.17)$$

The same expression is obtained when the granularity of goal management is increased by the scale factor g . That is, consider the total program execution time to be expressed as:

$$T = T_r + \frac{T_{gm}}{g} \quad (9.18)$$

The maximum speedup is expressed as:

$$S^{max} = \frac{T_r + \frac{T_{gm}}{g}}{T_r} = 1 + \frac{C}{g} \quad (9.19)$$

Therefore, whether we consider scaling down the execution time of goal reduction or scaling up the execution time of goal management, the effect on goal management complexity is the same. In the continued analysis, we consider only the scaling up of goal management and the same expressions apply to the scaling down of goal reduction.

The difference between the maximum possible speedup due to overlapped execution and the realistic speedup is determined by the incurred RU-GMU wait time. In the following analysis, we consider that the scaled execution time of goal management has the following two effects on the wait time. In the first case, we assume that the increased execution time is due to the increased number of goal management instructions, but their distribution is not changed. In this case, the realistic speedup due to overlapped execution is expressed as:

$$S^{r1} = \frac{T_r + \frac{T_{gm}}{g}}{T_r + W(g)} = \frac{1 + \frac{C}{g}}{1 + \frac{W(g)}{T_r}} \quad (9.20)$$

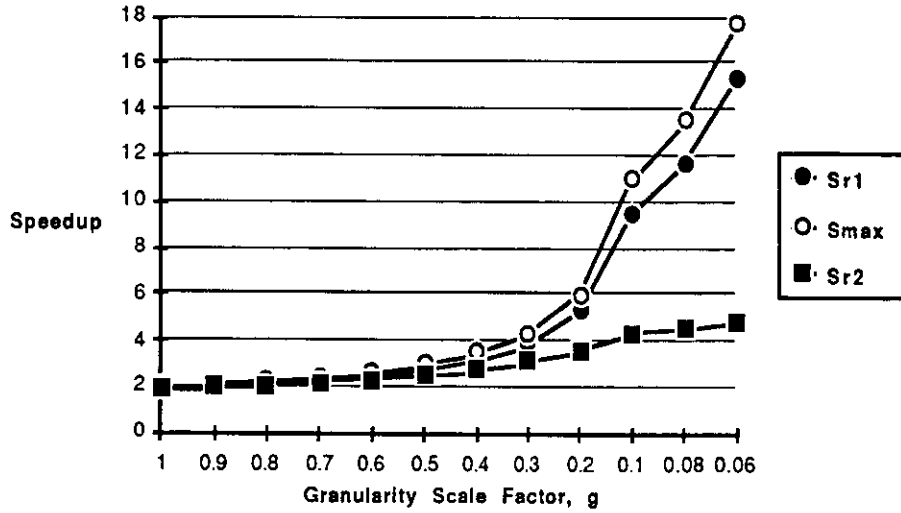


Figure 9.9: Speedup Versus Scaled Goal Management Complexity

In the second case, we assume that the scaled execution time of goal management affects the distribution of goal management instructions, so that the RU-GMU wait time is also scaled. In this situation, the speedup due to overlapped execution is expressed as:

$$S^{r2} = \frac{T_r + \frac{T_{gm}}{g}}{T_r + \frac{W(g)}{g}} = \frac{1 + \frac{C}{g}}{1 + \frac{W(g)}{T_r g}} \quad (9.21)$$

In Figure 9.9 we show the maximum speedup S^{max} , as well as the two modeled realistic speedups S^{r1} and S^{r2} . We observe the following results. Increasing the complexity of goal management, that is, reducing the granularity of goal reduction or increasing the granularity of goal management, results in higher wait times, due to the effect on the goal management instruction distance distributions. In other words, for the same number of goal reduction instructions and an increased number of goal management operations, the distribution will be more compact, resulting in more frequent delays. While the maximum possible benefit due to overlapped execution increases, the realistic speedup is dependent on the increase in the RU-GMU wait time.

In the case where the increased complexity of goal management results from more goal management instructions and not the goal management instruction distributions, S_{r1} , the realistic speedup due to overlapped execution of goal management follows quite closely the maximum possible value S^{max} . For example,


```

node(asleep,M,L,[msg(1,M)|S1],R):-
    node(active,M,L,S1,R?).
node(active,M,_L,[msg(2,I)|S1],[msg(1,I)|R1]):-
    M =\= I |
    node(active,M,I,S1,R1?).
node(active,M,_L,[],[msg(1,M)]).
node(active,M,L,[msg(1,L)|S1],[msg(2,J)|R1]):-
    L > J, L > M |
    node(active,L,L,S1,R1?).
node(active,M,L,S,[msg(2,_J)|R1]):-
    L < max(M,J) |
    node(passive,_,_,S,R1?).
node(passive,_,_,[Msg|S1],[Msg|R1]):-
    node(passive,_,_,S1,R1?).
node(passive,_,_,[],[]).

```

Figure 9.10: Program: Lord of the Rings

increasing the complexity by a factor of 10, results in an attainable speedup of close to 10, from a maximum value of 11.

We now consider a simple example of an FCP program that exhibits a higher goal management complexity than the System's Development Workload.

9.4.3 Modeling Communication Protocols

The “*Lord of the Rings*” algorithm computes the extreme value of nodes connected in a unidirectional circle using $O(N \log N)$ messages [Dole81]. The FCP program shown in Figure 9.10 is adapted from [Shaf84]. A unidirectional ring of N asynchronous *nodes* is created, with each node being in one of the following three states: *asleep*, *active* or *passive*. Each node contains a node identifier M , a *Send* and *Receive* communication channel to each neighboring node, and a local variable to store the identifier of the active process to its left. Initially, the created ring consists of nodes in the *asleep* state.

Without further describing the details of the algorithm, it is evident that each node performs very simple operations which include sending and receiving

Lord of the Rings, $N = 200$				
Creations	Terminations	Suspensions	Activations	Reductions
7342	7342	6779	6779	22745

Table 9.2: Goal Management Activity

messages on the input and output channels. The processing at each node consists of checking the state, and whether a message was received on the input channel. If a message is not present, the node suspends. Otherwise the message is received and as a result, a message is sent on the output channel.

Because of the simplicity of each node, the average goal reduction granularity is small, corresponding to the A-domain of program applications, as described in Figure 9.4. For example, the average granularity of approximately 30 FCP processor RISC instructions is obtained from a hand-compiled version of the program. To compute the goal management granularity, in Table 9.2 we show the total number of goal creations, terminations, suspensions and activations, as well as the total number of goal reduction, for a ring of $N = 200$.

The high suspension rate and goal management activity results in a goal management complexity that is greater than one ($C > 1$). For the optimized case where each goal suspension always suspends on only one variable (the receive channel), a goal management complexity of $C \approx 5$ is simulated.

CHAPTER 10

Thesis Summary, Conclusions and Future Work

In this thesis we propose a special-purpose single processor architecture for the efficient execution of the flat committed-choice logic programming language FCP. The processor provides architectural support for the main feature of the language, namely, the inter-goal communication, synchronization and, in general, goal management operations.

As part of the design procedure, which consists of empirical performance evaluation, we consider a real system workload that is representative of an existing environment used for system prototyping and development. We refer to this workload as the System's Development Workload. It consists of the Logix Operating System, the FCP Compiler, Debugger, Program Analyzer, and FCP Processor Simulator.

By analyzing the workload at the abstract machine level, we derive a set of algorithmic program parameters that do not depend on the physical machine implementation or the abstract machine emulation language. For example, the average number of variables that a goal suspends on is measured as 2.6 for the *System's Development Workload*. To derive the program parameters, a new instrumented version of the existing FCP abstract machine emulator was written, called *Statistics Logix* or *Slogix*. A detailed set of abstract machine characteristics can be found [Alka89].

The abstract machine parameters are used in the set of analytic performance models that describe the relative execution time of previously suspected implementation bottlenecks. The models are general, and applicable to other system workloads. Together, the individual models are combined to describe the average goal reduction execution time.

From the analysis, we conclude that the two main implementation bottlenecks are goal management and the overhead of redundant clause selection performed during *shallow backtracking*. The clause-selection overhead can be reduced using improved clause-indexing techniques. We model this improvement, and show

that this further results in an increase in the relative execution time of goal management. Therefore, the results of the analysis motivate the design of the special-purpose processor architecture, with architectural support for goal management. Goal management operations include goal creation, termination, suspension and activation.

The main feature of the special-purpose FCP single-processor architecture proposed in this thesis, is the multi-functional unit organization and its concurrent execution mechanism. The processor architecture is hierarchically structured in three layers, to provide a high memory bandwidth. In the first layer are the execution units: Reduction Unit, Tag Unit, Goal Management Unit and Instruction Unit. In the second layer are the following special-purpose cache units: Data and Tag Cache, Goal Cache and Instruction Cache. The third layer consists of the following specialized and dedicated memory modules: Data and Tag Memory, Goal Memory and Instruction Memory.

The most important feature of the FCP processor is the overlapped execution of goal management in the Goal Management Unit, and goal reduction in the Reduction Unit. The inter-unit communication protocol allows one overlapped goal management operation to be performed while the Reduction Unit continues program execution.

The efficient execution of goal management operations is performed using the special-purpose Goal Cache which is accessible by both the Goal Management Unit and the Reduction Unit. Fast goal switching, spawning and halting is performed by manipulating goal window pointers in the Goal Cache. Goal Cache *underflow* and *overflow*, as well as goal suspension and activation result in the manipulation of goal memory structures that are stored only in the Goal Memory. It is during this time that RU may have to wait, if another goal management operation is decoded, prior to the current one completing. For the System's Development Workload, we conclude that the overlapped execution of goal management operations, reduces the relative execution time of goal management from approximately 50% of the program execution time to approximately 3%. This is achieved using a Goal Memory bandwidth of 4 bytes/cycle.

The Data Cache in the FCP processor provides architectural support for data trailing, in the form of a special-purpose cache policy called *Delayed Binding*. Using this policy, all bindings performed during a clause-try are marked as temporary in the Data Cache. If the clause-try succeeds, the bindings are marked as permanent. However, if the clause-try fails or suspends, the bindings

are cleared. The relative execution time of data trailing in the FCP processor that does not have any special-purpose support is evaluated at 22%.

Combining the architectural support for goal management and data trailing in the Data Cache, we compared the FCP processor to the existing special-purpose processor proposed in [Hars88]. We show that the FCP processor provides a speedup of at least 2.5, for the same system workload.

The benefit of overlapped goal management is directly proportional to the relative execution time of goal management operations compared to the execution time of goal reduction operations. We refer to this ratio as the Goal Management Complexity, \mathcal{C} . The maximum speedup due to overlapped execution using the FCP processor is equal to $S^{max} = 1 + \mathcal{C}$. Since the System's Development Workload exhibits a goal management complexity of $\mathcal{C} \approx 1$, the maximum speedup due to overlapped execution is equal to 2.

Other application areas such as modeling distributed algorithms and communication protocols, exhibit a higher complexity of goal management, that is, $\mathcal{C} > 1$. For this application domain, the attainable speedup of overlapped goal management is higher than 2. We simulated the execution of the *Lord of the Rings* distributed algorithm for finding the extrema in a unidirectional ring of asynchronous nodes. A speedup of over 5 was measured due to overlapped execution of goal management, relative to a system where there is no architectural support for goal management.

10.1 Future Work: Shared Memory Multiprocessor

The research described in this thesis opened a number of interesting research issues that can be addressed in the immediate future. One such aspect of our work is a shared-memory multiprocessor architecture for the execution of FCP, which is composed of the defined special-purpose processors. Even though a shared-memory simulator for the multiprocessor architecture was developed, its analysis and evaluation was beyond the scope of this thesis. We now describe this architecture in more detail.

Since the model of computation of Concurrent Logic Programming languages is based on asynchronous communication between concurrent goals, the machine architectures considered by other researchers, for the execution of these class of languages, were generally based on distributed systems where inter-processor

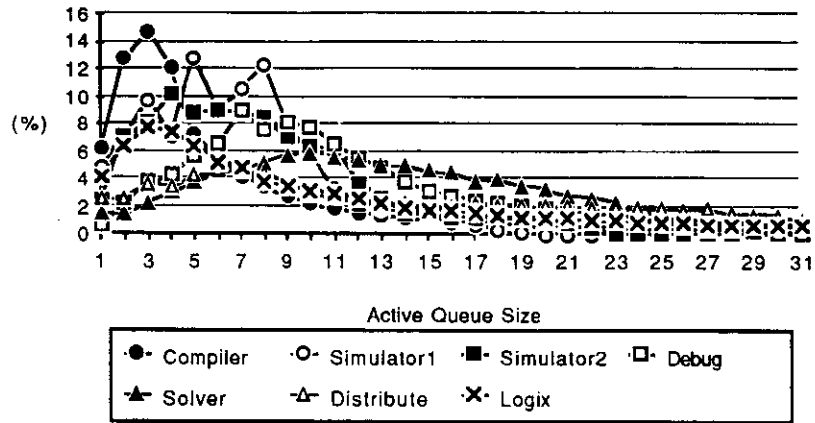


Figure 10.1: Distribution of the Active Goal Queue Size

communication is based on message-passing. However, our characterization of an existing Concurrent Logic Programming working environment, strongly suggests that the system of concurrent goals are tightly coupled. For small and medium scale multiprocessor organizations based on message-passing, the penalty of inter-process communication overcomes the benefit of parallel execution. This accounts for the performance degradation results obtained on the 16-node Hypercube, compared to sequential execution, [Tayl88].

Active Goal Queue Size

In order to get a *first impression* of the degree of concurrency available in the System Development Workload, we measured the size of the active queue during program execution. This distribution is shown in Figure 10.1, for the queue size values between 0 and 30. The results are surprising. It was generally thought that the size of the active goal queue would be very large. However, for all programs the maximum value of the distribution is for queue size values less than 12.

In Table 10.1 we show the maximum queue size, the average queue size value for each program, and the percentage of the queue size distribution not accounted for in Figure 10.1. Note that the *Compiler* program reached a maximum queue size value of 3195, whereas the other programs had values between 50 and 200.

Even though the maximum number of goals at some point can reach several

Size	FCP Benchmark Programs						
	Compiler	Sim1	Sim2	Debug	Solver	Distr.	Logix
Max.	3195	72	56	50	51	148	200
Ave.	6	7	8	11	13	10	7.5
Range	10%	0.2%	0.8%	1.2%	2.4%	24%	20.5%

Table 10.1: FCP Active Queue Size: Maximum and Average

thousand, the average size is much smaller. For the workload we consider, the average is approximately 10. On the other hand, the application programs were not selected for their concurrency, nor do they run on data that may result in a higher degree of concurrency. It would be of interest to consider such applications in the future.

If we assume that the granularity of the goals is of the same order of magnitude, then the distribution does indicate that the degree of parallelism is confined to a small to medium scale multiprocessor system. We now raise the following question: Can the implicit concurrency be exploited effectively in a multiprocessor system?

Shared-Memory Multiprocessor Organization

In Figure 10.2, we show the special-purpose multiprocessor organization. The separate Goal, Data and Instruction Memories are now shared via three special-purpose busses. The only modifications to the processor organization relative to the single processor architecture are the addition of standard features to implement mutual exclusion and access control to shared areas in memory. The most problematic part is the shared Data Memory, since it requires access control to shared logical variables. To reduce the potential overhead of contention for locks on variables, a separate *Lock Bus* and a *Lock Cache* per processor could be used.

The most interesting feature of the multiprocessor architecture, is the sharing of goals via the shared Goal Memory. We now discuss this aspect.

Goal Sharing in the FCP Multiprocessor

The Goal Management Unit, in conjunction with the Goal Cache allows the efficient execution of goal management operations. When the Goal Cache over-

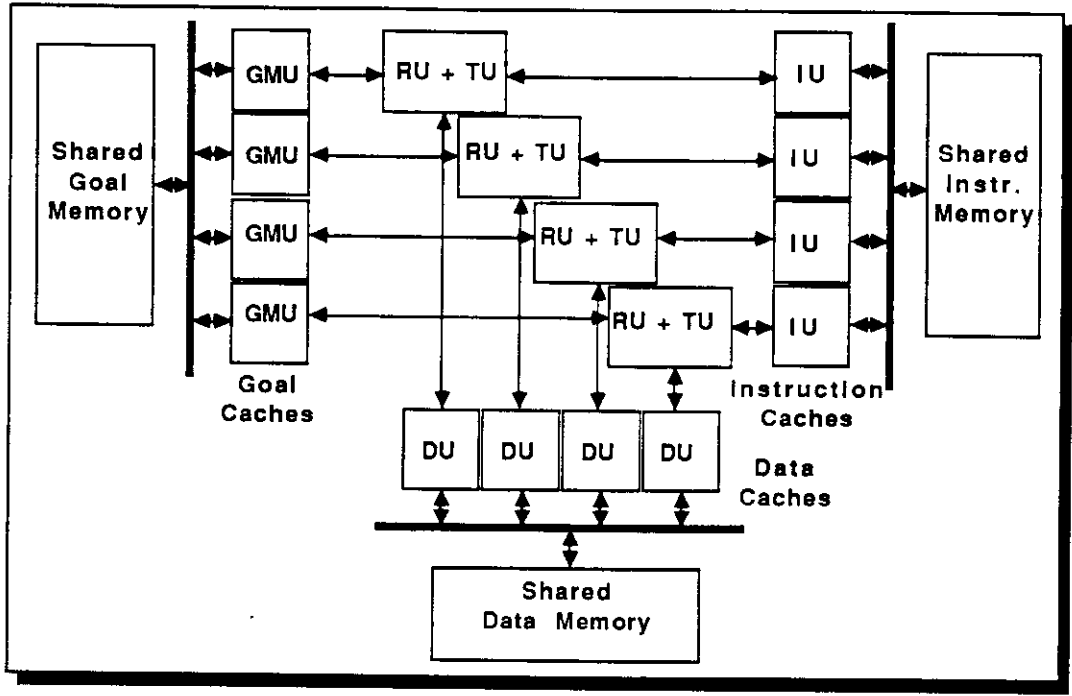


Figure 10.2: A Shared Memory FCP Processor Architecture

flows, it means that there are too many goals in the Goal Cache. A goal is then moved to the Goal Memory. In the multiprocessor configuration, the Goal Memory is shared. Therefore, an idle processor can *pick* up the goal that overflowed, and continue program execution. In other words, a goal may *migrate* from a busy processor to an idle processor. Moreover, these operations are performed while the Reduction unit is busy.

The size of the Goal Cache determines the tradeoff between the eagerness for sharing and the goal memory traffic that results from sharing. A more *eager sharing* system, for example, with a minimal cache size, is likely to result in a more balanced distribution of goals and a more utilized multiprocessor system. However, the tradeoff is the performance degradation due to contention for the shared Goal Memory. More traffic for the shared goal memory would slow down the goal management execution time, which would result in longer effective execution time of goal management operations (longer RU-GMU wait times), and thus reduce performance. The goal sharing approach is shown in Figure 10.3.

One thing should be noted. In most of the parallel implementation of Concurrent Logic Programming systems, the main problems that resulted in very poor performance, was the overhead of goal management and the poor load bal-

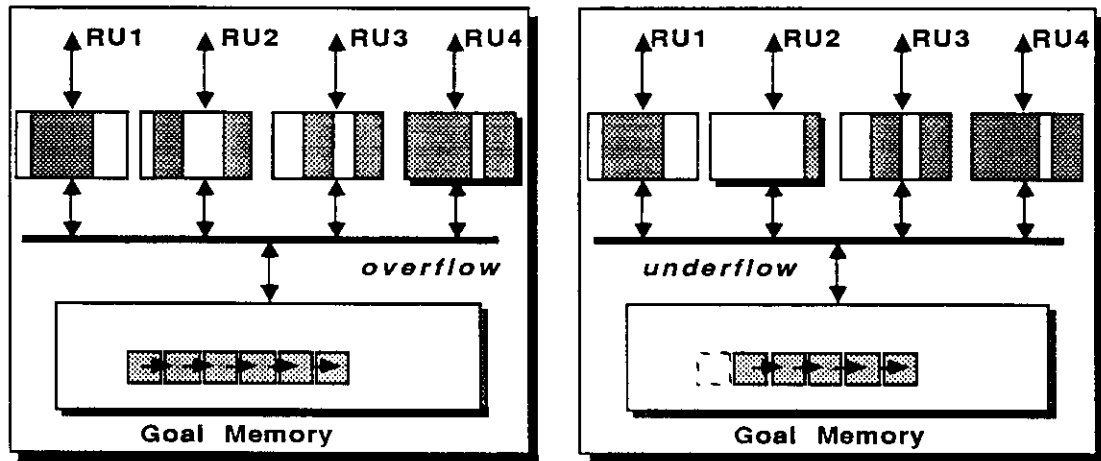


Figure 10.3: Load Balancing of Goals

ancing strategy. We propose a solution to the goal management problem by overlapping their execution using the Goal Management Unit. In addition, in a multiprocessing environment, the Goal Management Unit can also perform load balancing.

One additional feature of the Goal Management Unit could be to *listen* to the shared Goal Memory Bus, whenever it has no goals to schedule for execution, that is, whenever it is idle. As soon as one of the Goal Caches of the busy Reduction Units overflows, the first idle Goal Management Unit *takes* this goal without it being stored in the Goal Memory. In our immediate research, we plan to pursue these and other ideas.

Bibliography

- [Abe87] S. Abe, T. Bandoh, S. Yamaguchi, K. Kurosawa, and K. Kiriya, High Performance Integrated Prolog Processor, IPP, pp. 100 – 107, in *14th Annual Symposium on Computer Architecture* (June 1987).
- [Agha86] Gul Agha, *Actors, A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge Massachusettes (1986).
- [Alka88] L. Alkalaj and E. Shapiro, An Architectural Model for a Flat Concurrent Prolog Processor, pp. 1245 – 1323, in *Proceedings of the 5th International Conference/Symposium on Logic Programming* (Aug 1988).
- [Alka89] L. Alkalaj, *Flat Concurrent Prolog Abstract Machine Characteristics*, Technical Report CSD-890018, University of California, Los Angeles (April 1989).
- [Bar-86] U. Bar-on, *A Distributed Implementation of Flat Concurrent Prolog*, Master's Thesis CS 86, Weizmann Institute of Science, Applied Mathematics Department (January 1986).
- [Baro88a] U. Baron, J. Chassin de Kergommeaux, M. Hailperin, M. Ratcliffe, P. Robert, and J. Syre, The Parallel ECRC Prolog System PEPSys: An Overview And Evaluation Results, pp. 841–850, in *International Conference on Fifth Generation Computer Systems 1988* (November 1988).
- [Baro88b] U. Baron, B. Ing, M. Ratcliffe, and P. Robert, A Distributed Architecture for the PEPSys Parallel Logic Programming System, pp. 410 – 413, in *Proceedings of the 1988 International Conference on Parallel Processing, Vol. 1* (August 1988).
- [Bere87] A. Berenbaum, B. Colbry, D. Ditzel, R.D. Freeman, H. R. Mclellan, and K. J. O'Connor adn M. Shoji, CRISP: A Pipelined 32-bit Microprocessor with 13-kbit of Cache Memory, *IEEE Journal of Solid-State Circuits*, 22(5):776 – 782 (Octorber 1987).
- [Birr87] A. D. Birrell, J. V. Guttag, J. J. Hornig, and R. Levin, *Synchronization Primitives for a Multiprocessor: A formal Specification*, Technical Report 20, DEC Systems Research Center (August 1987).

- [Brat86] I. Bratko, *Prolog Programming for Artificial Intelligence*, Addison-Wesley (1986).
- [Chan85] J. H. Chang and A.M. Despain, Semi-intelligent backtracking of Prolog based on the Static Data Dependency Analysis, pp. 43 – 70, in *Proceedings of the IEEE Symposium on Logic Programming* (Aug 1985).
- [Ciep83] A. Ciepilewski and S. Haridi, A Formal Model for OR-Parallel Execution of Logic Programs, pp. 299 – 305, in *IFIP 83 Conference*, North Holland (1983).
- [Clar86] K. Clark, PARLOG: Parallel Programming In Logic, *ACM Trans. Prog. Lang. Syst.*, 8(1):1 – 49 (January 1986).
- [Cone87] John S. Conery, *Parallel Execution of Logic Programs*, Kluwer Academic Publishers (1987).
- [Cram86] Jim Crammond, An Execution Model for Committed-Choice Non-Deterministic Languages, pp. 148 – 158, in *1986 Symposium on Logic Programming* (September 1986).
- [Cram88] J. Crammond, *Implementation of Committed Choice Logic Languages on Shared Memory Multiprocessors*, Doctoral Dissertation PAR 88/4, Imperial College of Science and Technology (May 1988).
- [Dall88] W. J. Dally, *The J-Machine: System Support for Actors*, Massachusetts Institute of Technology, Cambridge, Massachusetts, Artificial Intelligence Laboratory and Laboratory for Computer Science (November 1988).
- [DeGr84] D. DeGroot, Restricted AND-Parallelism, pp. 471 – 478, in *Proceedings of the International Conference on Fifth Generation Computer Systems* (November 1984).
- [Dobr85] T. P. Dobry, A. M. Despain, and Y. N. Patt, Performance Studies of a Prolog Machine Architecture, pp. 180 – 190, in *12th Annual Symposium on Computer Architecture* (June 1985).
- [Dobr87] T. P. Dobry, *A High Performance Architecture for Prolog*, Doctoral Dissertation UCB/CSD 87/352, University of California, Berkeley (May 1987).
- [Dole81] D. Dolev, M. Klawe, and M. Rodeh, *An $O(N \log N)$ Unidirectional Distributed Algorithm for Extrema Finding in a Circle*, Technical Report RJ3185, IBM Research Laboratory, San Jose (July 1981).

- [Fagi87a] B. Fagin and A. M. Despain, Performance Studies of a Parallel Prolog Architecture, pp. 108 – 116, in *14th Annual Symposium on Computer Architecture* (June 1987).
- [Fagi87b] B. S. Fagin, *A Parallel Execution Model for Prolog*, Doctoral Dissertation UCB/CSD 87/380, University of California, Berkeley (November 1987).
- [Ferr78] Domenico Ferrari, *Computer Systems Performance Evaluation*, Prentice-Hall (1978).
- [Ferr89] Domenico Ferrari, Workload Characterization of Tightly-Coupled and Loosely-Coupled Systems, pp. 210, in *Proceedings of the 1989 ACM SIGMETRICS and PERFORMANCE 89* (May 1989).
- [Flyn88] M. Flynn, Foreword, in V. Milutinovic, editor, *High-Level Language Architectures*, pp. 1 – 2, North Holland (1988).
- [Fost87] I. Foster and S. Taylor, Flat Parlog: A Basis for Comparison, *International Journal of Parallel Programming*, 16(2): (1987).
- [Fuch86] K. Fuchi and K. Furukawa, The Role of Logic Programming in the Fifth Generation Project, pp. 1 – 24, in *Proceedings of the 3th International Conference on Logic Programming* (July 1986).
- [Gino87] R. Ginosar and A. Harsat, *Profiling LOGIX: A Step Towards a Flat Concurrent Prolog Processor*, EE Pub. Technical Report No. 617, Technion Institute of Technology, Haifa (January 1987).
- [Gold86] A. Goldberg, *Smalltalk*, Prentice-Hall (1986).
- [Goto87] A. Goto and S. Uchida, Towards a High Performance Inference Machine; The Intermediate Stage of PIM, in *Future Parallel Computers, LNCS 272*, pp. 299 – 320, Springer-Verlag (1987).
- [Hars88] A. Harsat and R. Ginosar, CARMEL-2: A Second Generation VLSI Architecture for Flat Concurrent Prolog, pp. 962–970, in *International Conference on Fifth Generation Computer Systems 1988* (November 1988).
- [Hell83] L. Hellerstein and E. Shapiro, *Implementing Parallel Algorithms in Concurrent Prolog: The MAXFLOW Experience*, Technical Report CS 83-12, Weizmann Institute of Science, Applied Mathematics Department (August 1983).
- [Hoar85] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall (1985).

- [Horo87] M. Horowitz, P. Chow and D. Stark, R. Simoni, A. Salz, S. Przybylski, J. Henessey, G. Culak and A. Agrawal, and J. M. Acken, MIPS-X: A 20 MIPS Peak 32-bit Microprocessor with On Chip Cache, *IEEE Journal of Solid-State Circuits*, 22(5):790 – 799 (October 1987).
- [Hour86] A. Hourri and E. Shapiro, *The Sequential Abstract Machine for Flat Concurrent Prolog*, Master's Thesis CS 86-20, Weizmann Institute of Science, Applied Mathematics Department (July 1986).
- [Ichi87] N. Ichiyoshi, A Distributed Implementation of Flat GHC on the Multi-PSI, pp. 257 – 275, in *Proceedings of the 4th International Conference on Logic Programming* (Aug 1987).
- [Inte89] Intel, N10, *IEEE Spectrum*, 1(1):1 – 1 (April 1989).
- [Ito85] N. Ito, H. Shimizu, E. Kuno A. Kishi, and K. Rukosawa, Dataflow Based Execution Mechanism of Parallel and Concurrent Prolog, *New Generation Computing*, 3(1):15 – 41 (February 1985).
- [Ito86] N. Ito, M. Sato, E. Kuno, and K. Rokusawa, The Architecture and Preliminary Evaluation Results of the Experimental Parallel Inference Machine PIM-D, pp. 149 – 156, in *13th Annual Symposium on Computer Architecture* (June 1986).
- [Kish85] M. Kishi, E. Kuno, K. Rokusawa, and N. Ito, *The Dataflow-based Parallel Inference Machine To Support Two Basic Languages in KL1*, ICOT, TR-114, Institute of Fifth Generation Computers (July 1985).
- [Kish86] M. Kishishita, J. Tanaka, T. Miyazaki, K. Taki, and T. Chikayama, *Distributed Implementation of FGHC; Towards the Realization of Multi-PSI System*, ICOT, TR-159, Institute of Fifth Generation Computers (March 1986).
- [Klig87] S. Klinger, *Towards a Native Code Compiler for Flat Concurrent Prolog*, Master's Thesis CS 87, Weizmann Institute of Science, Applied Mathematics Department (July 1987).
- [Klig88a] S. Klinger and E. Shapiro, A Decision Tree Compilation Algorithm for FCP(:,?), pp. 1315 – 1336, in *Proceedings of the 5th International Conference/Symposium on Logic Programming* (Aug 1988).
- [Klig88b] S. Klinger, E. Yardeni, K. Kahn, and E. Shapiro, The Language FCP(:,?), pp. 763–774, in *Proceedings of the Fifth Generation Computer Systems 1988* (December 1988).

- [Koik86] H. Koike and H. Tanaka, Fast Execution Mechanism of a Parallel Inference Engine PIE: Pipelined Goal Rewriting and Goal Multitasking, pp. 159 – 169, in *Logic Programming 1986, Proceedings of the 5th Conference, LNCS 264*, Springer-Verlag (1986).
- [Kowa79] Robert Kowalski, *Logic for Problem Solving*, North-Holland (1979).
- [Kumo86] K. Kumon, H. Masuzawa, A. Itashiki, K. Satoh, and Y. Sohma, *Kabu-Wake: A New Parallel Inference Method and its Evaluation*, ICOT, TR-150, Institute of Fifth Generation Computers (March 1986).
- [Lave83] Steven Lavernberg, *Computer Performance Modelling Handbook*, Academic Press (1983).
- [Lin88] Yow-Jian Lin and Vipin Kumar, An Execution Model for Exploiting AND-Parallelism in Logic Programs, *New Generation Computing*, 5(2):393 – 425 (February 1988).
- [Lloy84] J. W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag (1984).
- [Lusk88] E. Lusk, D.H.D. Warren, S. Haridi, and et al, The AURORA OR Parallel Prolog System, pp. 819–830, in *International Conference on Fifth Generation Computer Systems 1988* (November 1988).
- [Mele89] Charles Melear, The Design of the 88000 RISC Family, *IEEE Micro*, 9(2):26 – 38 (April 1989).
- [Mier85] C. Mierowsky, *The Design and Implementation of a Flat Concurrent Prolog*, Master's Thesis CS 85-09, Weizmann Institute of Science, Applied Mathematics Department (July 1985).
- [Moon85] D. A. Moon, Architecture of the Symbolics 3600, pp. 76 – 83, in *12th Annual Symposium on Computer Architecture* (June 1985).
- [Mura85a] K. Murakami, T. Kakuta, and R. Onai, *Architectures and Hardware Systems: Parallel Inference Machine and Knowledge Base Machine*, ICOT, TR-084, Institute of Fifth Generation Computers (1985).
- [Mura85b] K. Murakami, T. Kakuta, R. Onai, and N. Ito, Research on Parallel Machine Architecture for Fifth-Generation Computer Systems, *IEEE Computer*, 18(6):76 – 92 (June 1985).
- [Naka85] R. Nakazami, A. Konagaya, S. Habata, H. Shimazu, M. Umemura, M. Yamamoto, M. Yokota, and T. Chikayama, Design of a High Speed Prolog Machine (HPM), pp. 191 – 197, in *12th Annual Symposium on Computer Architecture* (June 1985).

- [Naka87] H. Nakashima and K. Nakajima, Hardware Architecture of the Sequential Inference Machine: PSI-II, pp. 104 – 113, in *Proceedings of the 4th Symposium on Logic Programming* (Aug 1987).
- [Onai85a] R. Onai, M. Aso, K. Masuda H. Shimizu, and A. Matsumoto, Architecture of a Reduction-Based Parallel Inference Machine: PIM-R, *New Generation Computing*, 3(2):197 – 228 (February 1985).
- [Onai85b] R. Onai, M. Aso, H. Shimizu, K. Masuda, and A. Matsumoto, *Architecture of a Reduction-Based Parallel Inference Machine: PIM-R*, ICOT, TR-105, Institute of Fifth Generation Computers (June 1985).
- [Onai85c] R. Onai, K. Masuda, H. Shimizu, A. Matsumoto, and M. Aso, *Architecture and Evaluation of a Reduction-Based Inference Machine: PIM-R*, ICOT, TR-138, Institute of Fifth Generation Computers (September 1985).
- [Robi65] J. A. Robinson, A Machine Oriented Logic Based on the Resolution Principle, *J. ACM*, 12(1):23 – 41 (January 1965).
- [Sara87] V. Saraswat, GHC: Operational Semantics, Problems and Relationship with CP(\downarrow , |), pp. 347 – 358, in *Proceedings of the IEEE Symposium on Logic Programming* (August 1987).
- [Sara89] V. Saraswat, *Concurrent Constraint Programming*, Doctoral Dissertation CMU, Carnegie Mellon University (April 1989).
- [Seo87] K. Seo and T. Yokota, A Processor for High-Performance Execution of Prolog Programs, pp. 261 – 274, in *VLSI 1987*, Ed. Carlo H. Sequin, North Holland (1987).
- [Shaf84] A. Shafir and E. Shapiro, *Distributed Programming in Concurrent Prolog*, Technical Report CS 84-02, Weizmann Institute of Science, Applied Mathematics Department (January 1984).
- [Shap83] E. Shapiro, *A Subset of Concurrent Prolog and its Interpreter*, ICOT, TR-003, Institute of Fifth Generation Computers (January 1983).
- [Shap84] Ehud Shapiro, *System Programming in Concurrent Prolog*, Technical Report CS84-01, Weizmann Institute of Science, Applied Mathematics Department (January 1984).
- [Shap86] Ehud Shapiro, Concurrent Prolog: A Progress Report, *IEEE Computer*, 18(6):76 – 92 (August 1986).
- [Shap87] Ehud Shapiro, editor, *Concurrent Prolog: Collected Papers, Vol1 and Vol2*, MIT Press (1987).

- [Sohm85] Y. Sohma, K. Satoh, K. Kumon, H. Masuzawa, and A. Itashiki, *A New Parallel Inference Mechanism Based on Sequential Processing*, ICOT, TM-0131, Institute of Fifth Generation Computers (July 1985).
- [Stee87] P. Steenkiste and J. Hennessey, Tags and Type Checking in LISP: Hardware and Software Approaches, pp. 50 – 59, in *Second International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1987).
- [Ster86] L. Sterling and E. Shapiro, *The Art of Prolog*, MIT Press (1986).
- [Svob76] Liba Svobodova, *Computer Performance Measurement and Evaluation Methods: Analysis and Applications*, Elsevier Computer Science Library (1976).
- [Tayl86] G. S. Taylor, P. N. Hillfinger, and P. N. Larus, Evaluation of the SPUR Lisp Architecture, pp. 444 – 452, in *13th Annual Symposium on Computer Architecture* (June 1986).
- [Tayl88] S. Taylor, R. Shapiro, and E. Shapiro, FCP: A Summary of Performance Results, pp. 1364 – 1373, in *The Third Conference on Hypercube Concurrent Computers and Applications* (January 1988).
- [Tayl89] S. Taylor, *Parallel Logic Programming Techniques*, Prentice Hall (1989).
- [Thac88] C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite Jr., Firefly: A Multiprocessor Workstation, *IEEE Transactions on Computers*, 37(8):909 – 920 (August 1988).
- [Tick85] E. Tick, *Prolog Memory Reference Behavior*, Technical Report CSL-TR-85-281, Computer Systems Laboratory, Stanford University (September 1985).
- [Tick87] E. Tick, *Studies in Prolog Architectures*, Technical Report CSL-TR-87-329, Computer Systems Laboratory, Stanford University (June 1987).
- [Tick88] E. Tick, *Performance of Parallel Logic Programming Architectures*, Technical Report TR-421, ICOT, Japan (September 1988).
- [Tosh87] T. Toshiaki, T. Maruyama, and H. Tanaka, A Preliminary Evaluation of a Parallel Inference Machine for Stream Parallel Languages, in *Logic Programming 1987, Proceedings of the 6th Conference, LNCS 315*, pp. 132 – 147, Springer-Verlag (1987).

- [Ueda86] K. Ueda, *Guarded Horn Clauses*, Doctoral Dissertation, University of Tokyo (March 1986).
- [Warr83] David H. D Warren, *An Abstract Prolog Instruction Set*, Technical Report 309, Artificial Intelligence Center. SRI International (January 1983).
- [Warr87] D. H. Warren, OR-Parallel Execution Models of Prolog, pp. 242 – 259, in *Proceedings of the International Joint Conference on Theory and Practice of Software Development, LNCS 250* (Aug 1987).
- [Whit85] Colin Whitby-Stevens, The Transputer, pp. 292 – 300, in *12th Annual Symposium on Computer Architecture* (June 1985).
- [Wise86] Michael J. Wise, *Prolog Multiprocessors*, Prentice-Hall (1986).