A PARALLEL ARCHITECTURE FOR SERIALIZABLE PRODUCTION SYSTEMS

by

JOSÉ NELSON AMARAL, B.E., M.E.

DISSERTATION

Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN
December 1994

A PARALLEL ARCHITECTURE FOR SERIALIZABLE PRODUCTION SYSTEMS

APPROVED BY

DISSERTATION COMMITTEE:

Supervisor: _____

_____

_____

_____

_____

Acknowledgments

I am deeply thankful to my advisor, Prof. Joydeep Ghosh, for many enlighting discussions, for coherent and sound advising, for his patience in carefully reviewing all my writings, and for his great availability. Prof. Ghosh and I developed an excellent working relationship in which I felt free to pursue my own research interests. At the same time I could count with his help and guidance whenever I felt I could not continue on my own. When I depart from Austin, I bring with me not only the academic knowledge but also an inspiring example to base the construction of good relationships with my own students.

I appreciated the fruitful discussions with Prof. Daniel Miranker, Prof. Sanqi Li, Prof. Ben Kuipers, and Prof. Vijaya Ramachandran when I needed help in conducting my research.

It would have greatly extended my stay in graduate school have I had to write the simulator from scratch. I am very thankful to Anurag Acharya of Carnegie Mellon University who graciously let me use the front-end of his own simulator. Anurag was also very helpful with many questions and provided some of the benchmarks used to test the architecture. I would also like to acknowledge Howard Owens of Microelectronics and Computer Technology Corporation (MCC) for tracking a difficult bug in the implementation of the simulator.

I was very fortunate to meet many helpful people in my passage through graduate school. Among the staff: Mellanie Gulick, Adela Baines, and Shiree Schade. My officemates that were willing to proofread pieces of my work, or to listen when I felt the need to verbalize explanations to problems in my research: Jeff Draper, Kagan Tumer, and our "adopted officemate" Guillaume Brat. A big

thanks to our volunteer system administrators: Alex Tomlinson, Jeff Draper, Thomas Callaway, and specially Kurt Bollacker for the numerous hours they spent to make sure that our systems were functional.

The financial support for my graduate studies came from Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), a Brazilian government agency for the development of science and technology and from Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS). At PUCRS I am specially thankful to Profs. Urbano Zilles, Dulcemar Coelho Lautert, Juarez Seagebin Correa, and Hernan Crosa Berni who always demonstrated great support for the continuation of my education.

And last, but definitely not least, a very big thanks to all my friends that made my living in Austin so interesting. I sure will miss y'all.

José Nelson Amaral

The University of Texas at Austin
December 1994

A PARALLEL ARCHITECTURE FOR SERIALIZABLE PRODUCTION SYSTEMS

Publication No. _____

José Nelson Amaral, Ph.D.
The University of Texas at Austin, 1994

Supervisor: Joydeep Ghosh

This dissertation introduces a new parallel architecture for implementing production systems. Key innovations include the elimination of global synchronization before each production firing and the overlapping between the matching and the select-act phases of Production Systems. These innovations are made possible by the use modern associative memory devices as control supporting structures. Allowing a production to fire before the matching of previous production firings is complete proved to be very efficient. The results produced by the architecture are proven to be correct according to the serializability criterion.

The development of a new benchmark addresses the lack of good benchmarks for the study of novel production system architectures. This benchmark allows for variations in the number of productions, database size, size of local data clusters, and ratio between local and global data.

A study of the production partition problem resulted in four different algorithms. These algorithms take into consideration processor workload balance, production interdependency, replication of data in memory, and reduction of

communication traffic. Experimental studies with a comprehensive event driven simulator indicate that the use of dynamic information from previous runs produces more successful algorithms.

A comparative study with a parallel architecture that does global synchronization before every production firing shows that both improvements, namely the elimination of global synchronization and the overlapping between matching, selecting and firing, are very effective in improving the performance of production systems. Further measurements indicate that only a modest amount of associative memory is needed for this architecture and that the use of a bus as an interconnection network does not constitute a bottleneck.

Finally, a multiple functional unit Rete Network is considered within each processor of the architecture. New synchronization problems appear when multiple tokens are concurrently propagated through the Rete Network. Two synchronizing buffers assure correct operation of the architecture. Performance evaluations through system simulation and through analytical modeling indicate that using a modest number of functional units in the Rete Network is cost effective, but the architecture clearly yields diminishing returns when tens of functional units are used.

Table of Contents

List of Figures

List of Tables

List of Abbreviations and Acronyms

BIN	Broadcasting Interconnection Network
BNB	Broadcasting Network Buffer
CAM	Content Addressable Memory
CE	Condition Elements
CTSP	Contemporaneous Traveling Salesperson Problem
DNT	Dependent on Network Transactions
FIC	Firing Instantiation Controller
FIFO	First-In First-Out
FIM	Firing Instantiation Memory
I/OP	Input Output Processor
IC	Instantiation Controller
IFE	Instantiation Firing Engine
INT	Independent of Network Transactions
IOB	In-Order Buffer
LHS	Left Hand Side
ME	Matching Engine
PC	Production Controller
PMM	Pending Matching Memory
PM	Production Memory
PRG	Production Relationship Graph
PS	Production System
RAM	Reference Addressable Memory
RHS	Right Hand Side

ROB	Re-Order Buffer
SD	Snooping Directory
TSP	Traveling Salesperson Problem
VLSI	Very Large Scale Integration
WME	Working Memory Element

List of Symbols

Chapter 3

R_{rw} — Read-write ratio.
R_i — i^{th} production.
$A(R_i)$ — Set of antecedents of production R_i.
$C(R_i)$ — Set of consequents of production R_i.
W_k — k^{th} Working Memory Element.
$T[W_k]$ — Type of Working Memory Element W_k.
$S_{A(R_i)}[W_k]$ — Sign of Working Memory Elements of type $T[W_k]$ in the antecedents of production R_i.
$S_{C(R_i)}[W_k]$ — Sign of Working Memory Elements of type $T[W_k]$ in the consequents of production R_i.
P_i — i^{th} processor in the parallel architecture.
$\mathcal{I}$ — A collection of production instantiation.

Chapter 4

K — A continent.
C_i — The i^{th} country in the continent.
$P(C_i)$ — Predecessor of country C_i in a continent tour.
$S(C_i)$ — Successor of country C_i in a continent tour.
$b(C_i, C_j)$ — Center of the border between consecutive countries C_i and C_j.
μ_c — Average number of cities per country.
σ_c — Standard deviation for number of cities per country.
$m(i)$ — Number of cities in the i^{th} country.
$C_{\pi(i)}$ — The i^{th} country to be visited in a tour.
$c_{i,\tau(j)}$ — The j^{th} city to be visited in the tour of the i^{th} country.
$d(ck, i, cl, j)$ — distance between i^{th} city of k^{th} country and j^{th} city of l^{th}

country.

Chapter 5

R	— Set of vertices in Production Relationship Graph (PRG).
R_i	— The i^{th} vertex in PRG. It represents production R_i.
E	— Set of all edges of PRG.
E_{ij}	— The edge connecting R_i and R_j in PRG.
$w(E_{ij})$	— Weight of edge E_{ij} of PRG.
S_k	— The k^{th} subset of edges of PRG.
$cc(R_i)$	— Cut cost of vertex R_i.
$W(R_i)$	— Cumulative weight of vertex R_i.
a	— Maximum number of antecedents in any production.
c	— Maximum number of consequents in any production.
N	— Number of productions in a production system program.
$F(R_i, S_k)$	— Fitness of vertex R_i to subset S_k.
n	— Number of processors in the architecture or number of partitions to be generated by a partition algorithm.
$\mathcal{L}(S_i)$	— Work load of subset S_i.
$\mathcal{B}(R_j)$	— Average number of beta tests performed for production R_j in previous runs.
$B(S_i)$	— Number of beta tests expected to be performed for productions in set S_i.

Chapter 8

m	— Number of β-units in the Rete Network.
$S_{Rete}(m)$	— Speedup of Rete Network with m β-units.
T_α	— Average amount of time spent in α-nodes.
$T_\beta(m)$	— Average amount of time spent in β-nodes in a Rete Network with m β-units.
β_i	— i^{th} β-node in a Rete Network.
$\mathcal{B}$	— Set of all β nodes in a Rete Network.

$\mathcal{S}(\beta_i)$ — Signal of the antecedent associated with β_i.

$\mathcal{C}(\beta_i, \beta_j)$ — Indicates possible conflict among tokens directed to β_i and β_j.

$\mathcal{T}$ — Set of all tokens processed in a Rete Network.

T_i — The i^{th} token processed in a β-node.

$\mathcal{A}(T_i)$ — Indicates the action of token T_i (add or delete).

h_j — Probability that an α-node produced a bulk of j tokens to deliver to β-node processing.

$H(z)$ — Generating function for the h_j.

$\overline{H}$ — First moment of h_j.

$\overline{H^2}$ — Second moment of h_j.

g_i — Probability that the processing of a token in a β-node produces i new tokens.

$G(z)$ — Generating function for the g_i.

$\overline{G}$ — First moment of g_i.

$\overline{G^2}$ — Second moment of g_i.

λ_β — Arrival rate at β-units.

μ_β — β-unit processing rate.

p_k — Probability that k tokens are present in the β portion of the Rete Network, including the tokens being processed.

$P(z)$ — Generating function for the p_k.

ρ_1 — Utilization rate for a single β-unit Rete Network.

ρ_m — Utilization rate for an m-β-unit Rete Network.

$\overline{N}(m)$ — Average number of β-tokens in a Rete Network with m β-units.

$\overline{T}(m)$ — Average time to process a β-token in a Rete Network with m β-units.

Chapter 1

Introduction

> *"... I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted."*
> (Alan Turing, 1950) [87] as quoted in [34].

When Alan Turing proposed a general purpose machine, the word *computer* was typically used to designate a person, usually a number crunching clerk, whose function was to perform computations. Although Turing envisioned the use of a general purpose machine to free clerks from their tedious tasks, his long term goal and motivation was the construction of a *thinking machine*. After many years of theoretical and practical work, including important contributions to the construction of code-breaking machines that were key in the victory of the allied forces in World War II, Alan Turing abandoned the field of computing machine construction, predicting that by the turn of the century humankind

would have constructed a machine that could be regarded as a *thinker* [87].

Much has changed since the days when Gödel showed the incompleteness of arithmetic and Alan Turing and Alonzo Church were pondering Hilbert's *Entscheidungs-problem*. Hilbert had asked "Can mathematics be complete, consistent and decidable?" [34, 35]. While Church used lambda-calculus to tackle the problem, Turing conceived of a theoretical device, now famous as a *Turing machine*. Speculating about the numerous possibilities brought about by such machines, Turing posed the question "Can machines think?" This very question widened and rekindled old philosophical indagations about the meanings of "thinking" and of "being intelligent" [12, 66]. Turing's question is still unanswered and it remains a distant goal of humankind either to construct a thinking machine or to prove that such a machine cannot be constructed. Computer scientists and knowledge experts are still developing techniques that could eventually lead to a thinking machine. Meanwhile, philosophers continue to ponder the possibility of deciding whether such machines are indeed thinking.

As Turing indicated, the meaning of words change over time and even more so in periods of fast technological advances. For instance, the ready availability of computers in the second half of this century has influenced models of thought [9]. The transformation of Turing's dream into a scientific discipline required a narrow view of intelligence [76]. Although still quite far from the goal of constructing a thinking machine, the study of Artificial Intelligence (AI) has progressed in two directions: "Pure" AI is concerned with knowledge representation and with the construction of machines that mimic human intelligence; "Applied" AI tries to construct artificial systems that perform tasks that are assumed to require human intelligence, but does not concern itself with the ways these tasks are accomplished [47].

Studies in applied AI have branched into many areas of research such as knowledge systems [31], genetic and evolutionary algorithms [24, 25, 36], fuzzy systems [94], neural networks [33], interactive agents [57, 68], and asynchronous organization [17], just to name some.

In the area of knowledge representation and processing, Production Systems are predominant structures. In such systems, knowledge is represented in the form of statements about a given knowledge domain in which the system operates. The collection of all statements form a knowledge base. A set of productions, also called rules, specify changes to this knowledge base according to its current state. The productions to be executed are selected by the inference engine and their actions alter the knowledge base accordingly. This dissertation addresses the problem of building a computer architecture that implements Production Systems.

Early experiments with Production Systems followed the pure branch of AI: the interest was centered on reproducing "simple" human activities in a computer to understand how these activities are performed in the human brain. Scientists chose to investigate activities that are easily performed by a four-year-old child, such as building a pile of blocks. Early results were disastrous because the scientists had failed to "teach" their machine basic notions of space and gravitation law. As a result, the machine would start building the pile by trying to place the top block first [65]. These early experiments led scientists to understand that a great amount of common-sense knowledge is involved even in the most simple tasks. It was also observed that knowledge based systems would be much easier to build in restricted knowledge domains.

This underestimation of the importance and complexity of *common-sense knowledge* reflects the fact that people tend to regard as simple and trivial knowl-

edge that has been acquired a long time in the past [57]. This phenomenon is observed at the individual level and at the species level. For instance, a concept that seems difficult for a freshman is regarded as elementary by the same student in his senior year. At the species level, in the end of the twentieth century concepts such as quantum mechanics, fractals, quarks and black holes are understood only by a selected group of scientists. On the other hand, children understand currently accepted concepts such as the shape of the Earth and its movement, while the best scientists of the fifteenth century rejected these very ideas.

The initial failure with seemingly easy tasks motivated scientists to look for activities confined to a controlled environment where "common-sense knowledge" would not play such an important role as in a child's playground. This search pointed to the activities of experts, such as medical diagnosis, chemical analysis, computer system configuration, and airline routing planning. To the surprise of some, many expert activities were very suitable to be codified into a small set of rules. Such realizations lead to the birth of the *expert system* or *knowledge based system* area of research. The developments in this area transformed what started as pure AI study into an applied one: today the interest is centered on the extraction of knowledge from an expert to reproduce the task in a computer.

When a knowledge based system is used as an expert system that manipulates a number of "rules," it is also called a *rule-based system.* Production systems are commonly used to implement rule-based systems. When the inference engine of a production system is used to derive new knowledge from a set of basic facts or axioms, the system is called a *theorem proving system.*

A production system inference engine is said to use *forward chaining* or

forward inference when the system starts in an initial state and executes enabled productions to move to the final state. A *backward chaining* system starts at the goal state and tries to find the possible conditions that resulted in that state. Backward systems are often used for theorem proving while forward systems are common in expert systems [67]. In this research we study production systems with forward inference engines.

The investigation of new computer architectures for Production Systems is motivated by the unique characteristics and resource needs of such systems. First, the execution of Production Systems requires an impressive amount of searching. Second, most of the data manipulated by Processing Systems is symbolic. Because of these characteristics, execution of Production Systems in general purpose computers have failed to deliver the speed required by many industrial and commercial activities.

Chapter 2 presents a generic architecture for Production System that emphasizes the matching engine. Most of the previous research in the area has concentrated on the matching phase of the system. The presentation of the Rete Network is followed by a discussion of this early work, as well as an analysis of current trends of research. A survey of some previous experimental work is given to show that, in production systems, data is accessed much more often than it is modified.

The presentation of a concurrent production system architecture in Chapter 3 is preceded by a number of important definitions. This novel architecture consists of a small number of powerful processors that implement a parallel production system without global synchronization. After providing a detailed description of the architecture and processing model, we present a proof that the proposed architecture produces correct results under the serializability criterion.

The absence of a comprehensive set of benchmarking tools is a well-known weakness of production system research. Chapter 4 presents a new benchmarking facility that is a modification of the Traveling Salesperson Problem. The great advantage of this benchmark is that it allows for variation in the number of productions, database size, ratio between local and global data, and size of local data clusters. Chapter 4 also describes and presents static measures for other performance evaluation benchmarks.

Chapter 5 studies the problem of partitioning productions among processors in a parallel machine. All the partition algorithms presented have a common set of goals: minimizing the duplication of working memory elements, reducing traffic in the bus, and balancing the amount of processing in each processor. Our experiments indicate that dynamic algorithms that consider the firing frequency of each production perform better than algorithms based exclusively on static information.

In Chapter 6, system simulation is used to measure performance. The most salient innovations in the new architecture are the elimination of global synchronization to resolve the conflict set and the overlapping between the matching and the select-act phases of production systems. The performance of the new architecture is compared to that of a synchronizing architecture that does not allow overlapping between production system phases. Other measures presented in this chapter include the following: measures for effectiveness of associative memories, estimates for memory size, and level of activity in the bus.

The evaluation of the architecture through system simulation in Chapter 6 indicated that the bottleneck in the architecture execution is in the processing of tokens in the β-nodes of the Rete Network. In Chapter 7 we extend the architecture by allowing the use of a multiple functional β-unit Rete Network

within each processor. Performance measurements indicate that considerable speedup is obtained with a modest number of additional functional units.

The ability to predict the benefits of a new device without constructing or simulating it, facilitates the process of designing a new computer architecture. Chapter 8 develops an analytical model based on queueing theory to predict the amount of performance improvement obtained in a Rete Network with m functional units.

The introduction of this novel architecture delivers performance improvements beyond what was thought possible just a few years ago. We hope that these results can inspire new research which will eventually lead to the construction of faster production system machines. A broader use of expert systems in commerce, industry, and services would release human experts from tedious activities and result in more efficient, safe, and cost-effective processes.

The slow advance since Turing's days indicates that progress towards the construction of a thinking machine might be counted in centuries rather than in years as Turing had anticipated. Moreover, because expert systems are brittle and only operate in a narrow area of structured knowledge, they have a very limited capability for knowledge representation, [32, 65]. Considered as a pure AI technique and viewed against the backdrop of the human enterprise to construct a thinking machine, production systems are just a "sliver" that might be forgotten in some fringe of history.

However, if production systems are viewed as an applied AI technique, they already have a successful history. The integration of databases with expert systems creates an ever increasing demand for faster machines that can execute production systems with large knowledge bases [31, 84]. Production Systems have also been integrated with other AI technologies to deliver adaptive support

systems [18]. Systems that for a long time seemed a mere academic curiosity such as medical diagnosers [10] are now delivering advisory services and reducing the cost of health care [14]. Expert systems have been replacing expensively trained human experts in many areas for more than fifteen years: a 1984 inventory reported over one-hundred thirty systems built [56] while a 1992 article list more than eighty vendors of tools to help design expert systems [84].

Chapter 2

Background

Considerable efforts have been made towards speeding up production system machines in the past twenty years [6, 52]. Originally, production systems were realized as interpreted language programs for sequential machines. The high cost of matching motivated the development of concurrent matching systems and, subsequently, systems that also allowed multiple productions to be fired at the same time. In a separate line of research, modern compile optimization techniques were developed to run production system programs more efficiently on general purpose sequential machines.

These efforts have led to great advances in the understanding of the issues involved in the construction of faster production system machines, but only limited improvement in actual performance. Also, there have been few attempts to integrate progress made in different areas: the use of the restrictive commutativity criterion for correctness and the notion of a match-select-act "cycle" forced even advanced architectures to perform synchronization before each production firing; compile optimization techniques were mostly restricted to sequential ma-

chines; many of the concurrent matching engines were constructed with a large number of small processors and were not combined with parallel firing techniques. Moreover, parallel processing researchers failed to take advantage of the fact that, in typical production systems, reading operations are performed much more often than writing ones.

2.1 A Generic Production System Architecture

Most of the research towards speeding up expert systems uses the architectural model represented in Figure 2.1. The memory of the system is divided into a set of productions or rules stored in the production memory, and a set of facts stored in the working memory. The working memory gets its name from the fact that it is used as a "scratch" memory where the system writes and overwrites partial results. Each fact of the knowledge domain is stored in this memory as a unit called a Working Memory Element (WME).

A production stored in the production memory consists of a set of conditions and a set of actions. In a forward chaining system, the productions are usually syntactically expressed with the conditions positioned to the left of an arrow. Therefore, the conditions are called the Left Hand Side (LHS) of the production. Similarly, the assertions or actions, positioned to the right of the arrow, are called the Right Hand Side (RHS) of the production. Some research groups have adopted a nomenclature that is adequate for both forward and backward chaining systems [16]. They label the conditions the *antecedents* of the production and the assertions the *consequents* of the production.

A production in the production memory is said to be *fireable* if all its non-negated conditions are satisfied and none of its negated conditions is satisfied.

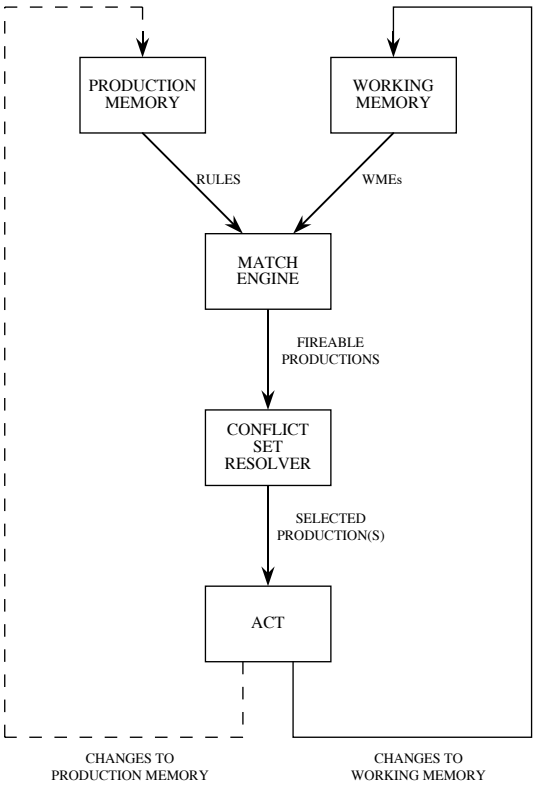

Figure 2.1: Generic Production System Architecture

Also, if variables appear in more than one condition element, all instantiations of the same variable must be bound to the same value. The match engine compares (or matches) all conditions of all productions in the production memory against all facts in the working memory, while keeping track of variable bindings to check which productions are fireable. The set of all fireable productions at the end of the match processing is called the *conflict set*. The conflict set resolver decides which production is selected to fire in the current cycle. Criteria used to select the winning production include: recency, specificity, priority, and context.

After a production is selected to fire, the act phase of the system produces changes in the memory, creating or deleting WMEs. Most production systems generate changes that affect only the Working Memory, altering the facts in the knowledge base. Some systems also generate new productions or eliminate old ones. We suggest that such systems be called *adaptive expert systems* [39] or *learning expert systems*, because they have the capability of adapting to changes in the environment.

2.2 The Matching Engine

The amount of work performed by the matching engine is a combinatorial function of the number of productions in the system and the number of facts in the knowledge base. However, two characteristics of production systems allow an efficient solution to this problem: distinct productions have identical condition elements and pieces of knowledge stored in the working memory of a production system change slowly over time. The slow change in the knowledge base implies that if the results of the matching in one cycle are saved for the next cycle, a substantial amount of work can be eliminated. The existence of productions with identical antecedents allows the construction of an algorithm in which the results of matching shared conditions are used by all productions that need it.

The best-known state saving algorithm is the Rete Network created by Forgy [20]. Forgy reports that Rete is inspired by the *Pandemonium* machine of Selfridge [74]. Pandemonium was one of the earliest learning machines and consisted of multiple layers of *demons*. A demon in a given level supervised an inferior level of demons. When it observed meaningful patterns, it sent messages to a superior level. The top-level demons performed more telling actions.

The Rete Network is a data-flow graph that encodes the antecedents of productions. The inputs to the Rete Network are changes to the working memory generated in the act phase of one cycle, and the outputs of the network are changes to the conflict set used to choose a production to be fired in the next cycle. The following discussion of Rete Networks is presented here after Gupta and Forgy [27, 29], Lee and Schor [53], and Miranker [58].

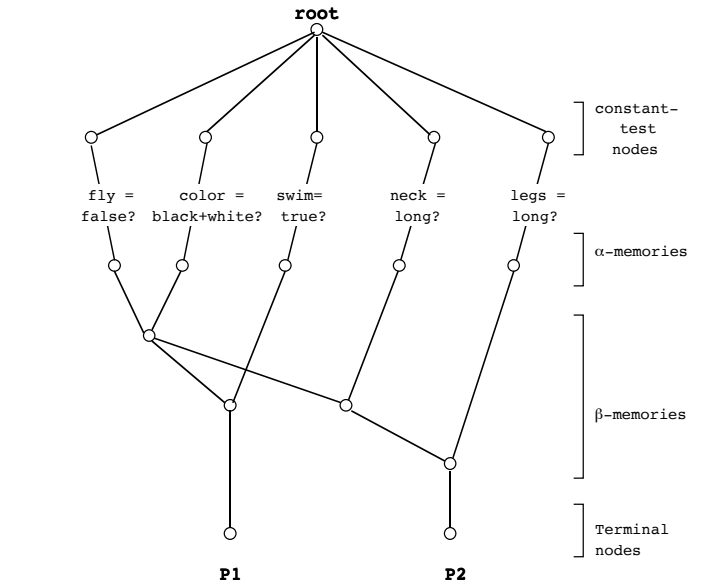

Figure 2.2: Section of a Rete Network

Figure 2.2 presents a Rete Network with the set of production antecedents encoded in it. The network is formed by four different kind of nodes: constant-

test nodes, memory nodes, two-input nodes and terminal nodes. The constant-test nodes appear in the first layer of the network. They store attributes that have constants in the value field and perform intra-condition tests to determine if a working memory element satisfies these constant fields of the condition element. In the original Rete Network, the result of this test was stored in a local α-memory [29]. Some authors claim that the α-memories can be eliminated because the constant test takes a negligible amount of time [53]. In this case the one-input nodes are called α-nodes and are memoryless.

Two-input nodes, also called *and*-nodes, *join*-nodes, or β-nodes, perform the matching between distinct condition elements. A β-node has one memory associated with each input. A token arriving in a β-node come either from another β-node or from an α-node. Whenever a new token arrives in one of a node's inputs, it is compared with all tokens present in the other side memory. Good hashing techniques are necessary to speed up the matching in the two-input nodes. Otherwise, it might be necessary to process long lists of tokens. [27].

Performance evaluation, modifications, and improvements of Rete are found in the works of Gupta [27, 29], Lee & Schor [53], Stolfo [80], Kelly and Seviora [42, 43], Gaudiot and Sohn [23, 78], and Barachini & Theuretzbacher [7]. Miranker proposed a useful modification to the traditional Rete Network in which intermediate memory nodes are eliminated. This modified Rete is called Treat [58].

A study by Gupta [27] motivated extensive research on concurrent matching. Gaudiot and Sohn [23, 78] proposed a data-driven machine with parallel matching. They advocated the suitability of data-flow machines for the processing of production systems. Kelly and Seviora [42, 43] proposed a multiprocessor archi-

tecture that supports comparison level partitioning and consists of a matching multiprocessor attached to a host computer. Rowe *et al.* [8, 69] developed METE/PIPER, an extension of Rete to explore intra-production parallelism in match processing and conflict resolution. Other noteworthy parallel implementations of Rete appear in [3, 30, 39, 52, 80, 81, 73].

2.3 Current Research and Trends

Parallel firing systems is a current research topic that The ability to fire productions in parallel introduces some new issues such as the identification of dependencies among productions. Kuo and Moldovan [50, 51] Schmolze [71], Kuo *et al.* [49], and Xu and Hwang [92] have worked in this area and suggested solutions to ensure the correctness of a parallel firing engine operation. The partitioning of production among processors also becomes an important issue in parallel firing systems. Oflazer [64], and Xu and Hwang [92] presented some interesting results in this area. In most of the research geared towards parallel firing of productions, it is assumed that there is a synchronization point before each production fires, that is, no production firing takes place until all matching activities are completed.

There is no agreement on a good criterion to establish correctness of a parallel production system. Some researchers claim that serializability imposes too heavy a burden on the programmer, and suggest the use of a commutativity criterion that restricts the available parallelism even more [38]. Other researchers suggest that the serializability criterion is already too restrictive and argue the use of control structures tailored to each problem [62, 63]. A very recent work by Schmolze proposes the concept of a convergent production

set in which serializability provides complete control of the final result. In such a system serializability is certainly enough to guarantee correctness [72]. An study of serializability as a correctness criterion based on a transaction model for parallel rule firing can be found in Srivastava et al. [79]. There is yet a third school of thought that advocates a meta-rule mechanism to allow the designer of the system to specify explicitly the control structure [83].

The elimination of synchronization in parallel production systems follows a trend of research in other areas of systems architecture, such as the development of A-Teams and scale-efficient organizations [17, 60, 85]. We anticipate that some results from these areas might be useful in improving execution of production systems. In this research we wager that the serializability criterion is a winner.

Acharya and Tambe have obtained impressive speed increases in the matching phase of production systems by means of storing collections of WMEs instead of single WMEs in the Rete Network [2]. Gordin and Pasik explored the use of set-oriented constructs as a means to integrate database management systems (DBMS) and rule-based systems [26]. Wu and Browne proposed the use of set-oriented constructions in an object-based model for parallel rule-based systems, their goal is to increase concurrency [91]. An expansion on the use of collections to construct a parallel firing, asynchronous, collection-oriented system is still to be further studied. It is certainly a promising idea.

Another important area of research is in Production Systems languages and design aid tools. Murthy [60] correctly points out that for some problems, it is possible to gather a few hundred heuristics and put together a production system in a few weeks. However, maintaining these systems during their lifetimes might be very expensive. One example is the R1 system at DEC that at one time

required a few hundred people for its maintenance. Good software engineering techniques need to be adopted in the construction of production systems to avoid such situations.

2.4 Read-Write Ratio in Production Systems

This section analyzes some published research to determine the expected read-write ratio in production systems. We are interested in learning the number of WMEs read for each WME modification performed. This ratio is important in evaluating the possibility of using a broadcasting network as an interconnection mechanism in a multiprocessor production system machine.

In a comparative study between Rete and Treat, Nayak *et al.* [61] measures the number of productions, the number of condition elements per production, the number of PS cycles executed, and the number of addition and deletion of WMEs reproduced on Table 2.1. The benchmarks in this table are the Eight Puzzle (EP), the Missionaries and Cannibals (M&C), R1 Soar, and Neomycin-Soar (NM). EP and M&C are toy programs, R1-Soar is an implementation of a subset of the system R1 that configures computer systems for DEC, implemented in Soar, and Neomycin-Soar is an implementation of a portion of Neomycin in Soar. Neomycin diagnoses infectious diseases like meningitis. The authors call attention to the fact that in a Soar program there is an average of 9 condition elements per production, compared with an average of 3 in OPS5 programs.

We examine these results to estimate how many reads are there for each write. In other words, how many antecedents are tested for each action taken on the consequents of a production. Therefore we define the *Read-Write Ratio* R_{rw}, as follows:

Benchmark	# Prod.	$\frac{CEs}{Prod}$	# Cycles	Add.	Del.
EP	71	8.8	796	1,715	1,313
M&C	78	8.9	1,617	6,609	5,620
R1-Soar	334	9.9	3,732	7,810	6,458
NM	419	8.8	2,173	4,554	4,336

Table 2.1: Measures by Nayak *et al.*

$$R_{rw} = \frac{\#\,reads}{\#\,writes} = \frac{\#\,WMEs\,probed}{\#\,WMEs\,added + \#\,WMEs\,deleted} \tag{2.1}$$

Table 2.2 shows numbers computed based on the results published by Nayak *et al.* The total number of writes in the first column is just the sum of the number of WMEs written and the number of WMEs read. If changes to the Working Memory use modify operations (like in OPS5) the number of writes tend to be smaller. If no modify operation is available, a modification needs to be implemented by a delete followed by an add operation. The total number of Condition Elements in the program was computed just by multiplying the average number of CEs per production by the number of productions. We assume that the conflict set is generated from scratch every time, therefore the total number of reads is equal the total number of CEs multiplied by the number of cycles that takes for the system to reach a state with the solution. This implies a very optimistic scenario where each WME is read only once in each cycle. The truly worst case has a complexity that is exponential in the number of antecedents in a production. The ratio R_{rw} varies between 100 and 1000, and is bigger for more "real life" problems. This number would be significantly smaller if the entire conflict set would not be generated before firing each production, or if some form of state saving is used. Even with such improvements, reads are

expected to significantly outnumber writes.

Benchmark	Total # Writes	Total # CEs	Total # Reads	R_{rw}
EP	3,028	625	497,500	164
M&C	12,229	694	1,122,198	92
R1-Soar	14,268	3,307	12,341,724	864
NM	8,890	3,687	8,011,851	901

Table 2.2: Read-Write ratio (R_{rw}) estimates based on the study by Nayak *et al.*

Perhaps the most important measurement study on production systems to date was published as the Ph.D. dissertation of Anoop Gupta [28]. He measured six production systems: R1, a program for configuring VAX computer systems; XSEL, a program which acts as a sales assistant for VAX computer systems; PTRANS, a program for factory management; HAUNT, an adventure game program; DAA, a program for VLSI design; and SOAR, an experimental problem solving architecture implemented as a production system. Gupta calls "surface measurement" the measures that one can do using exclusively the text of a production system without concern with the initial data base. Table 2.3 presents some of the surface measurements done by Gupta, as well as the R_{rw} computed by us.

Gupta presents the average number of actions and conditions per production, we compute the R_{rw} based on these numbers. Once more we assume that a conflict set is generated from scratch at every cycle, therefore the number of reads is given by the product of the number of productions and the average number of CEs per production. This measure is expected to be smaller in a parallel production system due to the partial decoupling among group of productions assigned to distinct processors. Also if the conflict set is not generated

Benchmark	# Prod	$\frac{CEs}{Prod}$	$\frac{Act}{Prod}$	# types	$\frac{Attr}{CE}$	$\frac{Vars}{CE}$	R_{rw}
R1	1,932	5.58	2.90	31	4.73	1.61	3,717
XSEL	1,443	3.84	2.41	36	3.64	0.96	2,299
PTRANS	1,016	3.12	3.64	81	4.11	2.14	871
HAUNT	834	2.41	2.51	23	2.08	0.24	801
DAA	131	3.91	2.86	20	3.89	2.69	179
SOAR	103	5.80	1.83	12	3.78	1.70	326

Table 2.3: Surface measurements by Gupta and the corresponding R_{rw} estimates.

at every cycle, or if some state saving algorithm is used, the number of reads will be smaller. Gupta measures another interesting number, that is, the number of WME types in each benchmark (showed in Table 2.3 as number of types). This gives us an idea of the number of different "types" or classes of productions in the expert system and indicates the potential to partition the productions into independent sets.

The analysis of Nayak *et al.* and Gupta's experimental work confirms that the number of times that the knowledge base is probed is much higher than the number of times that it is modified. Such a situation suggests that a parallel architecture based on a broadcasting network with concurrent reading and exclusive writing operations might be an efficient design.

Chapter 3

Architectural Model

"Everything should be made as simple as possible, but not simpler."
(Albert Einsten) as quoted in [57].

The architectural model proposed in this research consists of a moderate number of processors interconnected through a broadcasting network. The set of productions is partitioned among these processors with each production assigned to exactly one processor. A processor reads data only from its local memory, i.e., no read operations are performed over the network. Due to the absence of network reads and the low frequency of network writes, a simple bus should be adequate as the broadcasting system. This conclusion is supported by detailed experimental results showing the bus not to be a bottleneck even for a twenty processor system. A number of associative memories implement a system of lookaside tables to allow parallel operations in each processor. This scheme does not allow parallel production firing within a processor, but allows the match-

select-act phases of a PS to overlap. A snooping directory isolates the activities in remote processors from the activities in a local processor, and interrupts a local operation only when pieces of data that affect the local processor are broadcast over the network.

Section 3.1 presents basic definitions that set the environment for the processing model. Section 3.2 introduces the architectural organization and expands on the processor model, conflict set management, and processor operation. Section 3.3 presents a theorem that demonstrates that the results produced by the processing model is correct according to the serializability criterion of correctness.

3.1 Basic Definitions

A Production R_i consists of a set of antecedents $A(R_i)$ and a set of consequents $C(R_i)$: the antecedents specify the conditions upon which the production can be fired; the consequents specify the actions performed when the production is fired.

Definition 3.1 *The database manipulated by a Production System consists of a set of assertions. Each assertion is represented by a* **Working Memory Element** *(WME), notated by W_k. A WME consists of a class name and a set of attribute-value pairs that together characterize its* **type**, *$T[W_k]$.*

Consequence 3.1 *Two WMEs of the same type are distinguished only by the values associated with their attributes.*

Definition 3.2 *Each production antecedent specifies a type of WME and a set of values for its attribute-value pairs. A WME W_k is* **tested** *by an antecedent*

if it has the specified type. An antecedent is **matched** *by a WME if the WME has the type specified* and *all the values in the antecedent match the ones in the WME.*

Definition 3.3 *If the antecedents of a production R_i test WMEs of type $T[W_k]$, then W_k* **belongs** *to the antecedents of R_i, it is notated by $W_k \in A(R_i)$.*

Consequence 3.2 *A WME might belong to the antecedents of more than one production.*

Definition 3.4 *A* **non-negated antecedent** *tests for the presence of a WME in the memory. A* **negated antecedent** *tests for the absence of any matching WME in the memory. A production R_i is said to be* **fireable** *if all its non-negated antecedents are matched and none of its negated antecedents are matched.*

The consequent of a production can specify three kinds of actions that modify WMEs: addition, deletion, or modification.

Definition 3.5 *A WME W_k* **belongs** *to the consequents of a production R_i iff the firing of R_i adds (deletes) any WME of type $T[W_k]$ to (from) the Working Memory. This is denoted by $W_k \in C(R_i)$.*

Consequence 3.3 *A WME might belong to the consequents of more than one production.*

Definition 3.6 *If an antecedent of production R_i tests for the presence of a WME W_k, this is a* **positive** *test, notated by $S_{A(R_i)}[W_k] = +$, which is read as*

"R_i has at least one antecedent that tests for the presence of a WME of type $T[W_k]$". In a similar fashion, if the test is for absence of W_k, it is a **negative** *test, denoted by $S_{A(R_i)}[W_k] = -$.*

Definition 3.7 *When the consequent of a production specifies the addition of a WME W_k to Working Memory, it is a* **positive** *action, denoted by $S_{C(R_i)}[W_k] = +$. A* **negative** *action specifies the deletion of a WME W_k, denoted by $S_{C(R_i)}[W_k] = -$.*

Consequence 3.4 *The notation $S_{A(R_i)}[W_k] \neq S_{C(R_j)}[W_l]$ implies that $W_k \in A(R_i)$, $W_l \in C(R_j)$, and that R_i test of W_k is positive (negative) while R_j action on W_l is negative (positive).*

Consequence 3.5 *A modify action is considered as a combination of two actions: a negative one that removes the old WME and a positive one that creates the modified WME.*

In this model, productions are partitioned into disjoint sets with one set assigned to each processor. $R_n \in P_i$ indicates that production R_n *belongs* to processor P_i. The Working Memory is distributed among the processors in such a way that a processor stores in its local memory all WMEs tested by its productions. This is stated in Axiom 3.1.

Axiom 3.1 (Condition for Ownership) *A WME W_k is stored in the local memory of a processor P_i iff $W_k \in A(R_n)$ and $R_n \in P_i$.*

In the processing model discussed in section 3.2 some productions fire locally while others need to change WMEs that are stored in the local memory of remote

processors. The following definitions describe important situations that appear in the execution of the model.

Definition 3.8 *A WME W_k is* **local** *to a processor P_i iff W_k is stored in the local memory of P_i; W_k is not stored in the local memory of any other processor P_j; and there is no production allocated to a processor other than P_i that changes W_k.*

Definition 3.9 *A WME W_k is* **pseudo-local** *to a processor P_i iff W_k is stored in the local memory of P_i; W_k is not stored in the local memory of any other processor P_j; and there is at least one production allocated to $P_j \neq P_i$ that changes W_k. We say that P_j* **shares** *W_k.*

For example, a WME that is written by many processors and read by only one processor is *pseudo-local* for the processor that reads it; it is a *shared* WME for all processors that write it[1]. A processor does not stores shared WMEs in its local memory.

Definition 3.10 *A production R_n* **fires locally** *in a processor P_i iff $\forall W_k \in C(R_n)$, W_k is local or pseudo-local to P_i.*

Consequence 3.6 *A production that does not fire locally, is said to be a* **global production**. *Such a production must propagate actions to remote processors.*

Definition 3.11 *A production R_n* **enables** *a production R_m iff $\exists W_k$ such that $S_{C(R_n)}[W_k] = S_{A(R_m)}[W_k]$. A production R_n* **disables** *a production R_m iff $\exists W_k$ such that $S_{C(R_n)}[W_k] \neq S_{A(R_m)}[W_k]$.*

[1] It is also called a *shared action* or a *shared output*.

Definition 3.12 *A production R_n has an* **output conflict** *with a production R_m iff $\exists W_k$ such $S_{C(R_n)}[W_k] \neq S_{C(R_m)}[W_k]$.*

Productions that can fire locally are classified as Independent of Network Transactions (INT) or Dependent on Network Transactions (DNT), according to their dependencies with other productions that belong to other processors. INT and DNT productions have to be processed differently for correct execution according to the serializable criterion. The following definition states that a production that can fire locally is DNT if and only if any of the following conditions hold:

(i) two of the antecedents of the production are changed by the consequents of a single production allocated to another processor: one of these changes produces an enabling dependency and the other produces a disabling one;

(ii) the production has two conflicting writes with a production allocated to another processor;

(iii) the production has an output conflict and a disabling dependency with a production allocated to another processor.

Definition 3.13 *A production $R_n \in P_i$ is* **DNT** *iff R_n can fire locally and any of the following conditions hold:*

(i). *$\exists W_k, W_l$, and $R_m \in P_j \neq P_i$ such that W_k and W_l are pseudo-local for P_i, $S_{A(R_n)}[W_k] \neq S_{C(R_m)}[W_k]$, and $S_{A(R_n)}[W_l] = S_{C(R_m)}[W_l]$.*

(ii). *$\exists W_k, W_l$, and $R_m \in P_j \neq P_i$ such that W_k and W_l are pseudo-local for P_i, $S_{C(R_n)}[W_k] \neq S_{C(R_m)}[W_k]$, and $S_{C(R_n)}[W_l] \neq S_{C(R_m)}[W_l]$.*

(iii). $\exists W_k, W_l$, *and* $R_m \in P_j \neq P_i$ *such that* W_k *and* W_l *are pseudo-local for* P_i, $S_{A(R_n)}[W_k] \neq S_{C(R_m)}[W_k]$, *and* $S_{C(R_n)}[W_l] \neq S_{C(R_m)}[W_l]$.

Definition 3.14 *A production* R_n *is* **INT** *iff* R_n *can fire locally and* R_n *is not* **DNT**.

An INT production can start firing at any time as long as its antecedents are satisfied. A DNT production P_i only starts firing after all tokens generated by a production P_j, currently being fired by a remote processor, are broadcast in the network and consumed by the processor that fires P_i. This prevents P_i and P_j actions from being intermingled, avoiding thus non-serializable behavior.

3.2 System Overview

The parallel architecture presented in this dissertation stems from the realization that improvements restricted to the matching phase of the traditional match-select-act cycle of Production Systems (PS) fail to produce significant speedup. Even machines that allow concurrent execution of the acting and matching phases, while maintaining the global production selection, yield limited improvements in speed. The architecture proposed here allows parallel firing of productions allocated to distinct processors. Within a processor, activities related to matching, acting and selecting are concurrent. Thus the next instantiation to be fired may be selected even before the Rete Network updates due to a previous production firing are completed.

Such aggressive parallelism is possible because the concept of a match-select-act cycle is eliminated. The principle of firing the most recent and specific instantiation is replaced by an approximation of it: only instantiations that are

known at the time of the selection are considered, we call this a *partially informed* selection mechanism. The use of associative memories allows for quick elimination of instantiations that are no longer fireable. We also replace the restrictive commutativity criterion by the serializability criterion of correctness. The use of serializability reduces the number of situations in which synchronization is necessary, increasing the amount of parallelism available.

The observation that in production systems reading operations are much more frequent than writing operations motivates an architecture based on a broadcasting network over which only writing operations occur. Such an architectural model imposes limits to the number of processors used. However, two characteristics of PS make them compatible with an architecture with a moderate number of processors: the amount of inter-production parallelism is limited and, as a PS grows, the size of the database grows much faster than its production set.

The parallel architecture is formed by identical processors connected via a Broadcasting Interconnection Network (BIN), as shown on Figure 3.1. At start-up the I/O processor (I/OP) loads the productions on all processors. System level I/O and user interface are also handled through the I/OP. The main components of each processor are the Snooping Directory (SD), the Matching Engine (ME), the Production Controller (PC) and the Instantiation Controller (IC). The Snooping Directory is an associative memory that identifies whether a token broadcast on the BIN conveys an action relevant to the local processor. Relevant tokens are kept in a Broadcasting Network Buffer (BNB) until the IC and the ME are able to process it. The Matching Engine is a Rete-based matcher that implements a state-saving algorithm. The IC uses specialized memory structures to maintain and rapidly update the list of fireable instanti-

ations. To perform this task, it has to monitor the outputs of ME as well as the firing of local (through PC) and global (through SD) productions. One of the memories controlled by IC is the Firing Instantiation Memory (FIM) that keeps a list of all the production instantiations that are enabled to fire. The Production Controller (PC) selects an instantiation to be fired from the list maintained by IC, and, whenever necessary, synchronizes the production firing with BIN operations to guarantee that production firings appear to be atomic.

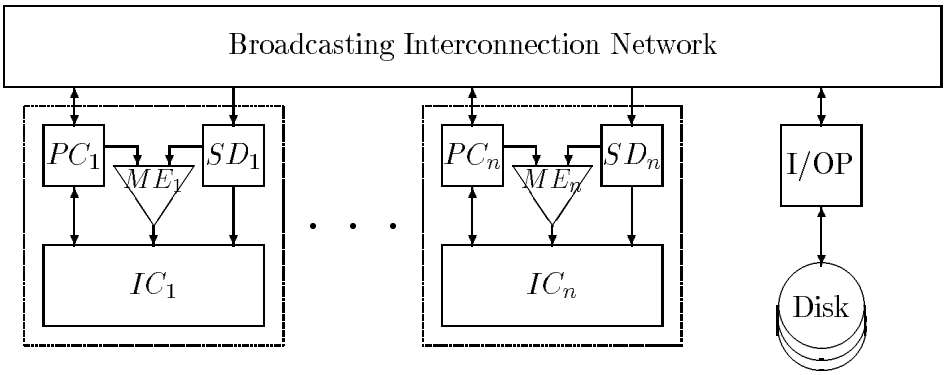

Figure 3.1: Parallel Machine Model

Productions are divided in three categories: local INT, local DNT, and global. The firing of a local INT production does not require BIN ownership because all its actions modify local WMEs only. Therefore, upon selecting an INT production, the PC immediately propagates its actions to ME and IC. To avoid interleaving of actions belonging to distinct productions, all tokens broadcast in BIN during local production firing are buffered in BNB. These tokens are processed as soon as the local firing finishes. When a local DNT production is selected, its execution has to wait until the BIN changes ownership, which is an indication that the firing of a global production has been concluded. The local DNT production is then fired in the same fashion as a local INT.

A global production modifies shared WMEs, i.e., WMEs that belong to the antecedents of productions assigned to other processors. Thus, these changes need to be broadcast to all processors. When a global production is selected, PC acquires access to the BIN, process all outstanding changes in the BNB, and, if the selected production is still fireable, proceeds to broadcast tokens with changes to shared WMEs. The BIN ownership is not released until all actions that change shared WMEs are broadcast. After releasing the BIN, PC prevents any incoming token from proceeding to local processing. These tokens are buffered in BNB and processed locally after the local execution of the selected production is complete. This avoids write interleaving in the local memories and guarantees an atomic operation for production firing within a processor.

The main steps in the machine operation are presented below in an algorithmic form. The steps of the algorithm are performed by different structures of the processing element.

Note that no production is fired while there are outstanding tokens in BNB. The selection of a fireable instantiation in step 2 of PRODUCTION-FIRING is done according to the "pseudo-recency" criterion: the most recent instantiation in FIM is selected[2]. This is not a true recency criterion because ME may still be processing a previous token, and thus the instantiations that it will produce are not in FIM yet.

[2]A single associative search in FIM produces the entry with the most recent time stamp.

PRODUCTION-FIRING

1. **execute** all outstanding tokens in BNB on first-come first-serve basis
2. **select** a fireable instantiation I_k in FIM
3. **if** I_k is global
4. **then** Request BIN ownership
5. **while** BIN ownership is not granted
6. **execute** tokens broadcast in BIN captured by SD
7. **if** I_k is still fireable
8. **then broadcast** actions that change shared WMEs
9. **execute** actions that change shared WMEs
10. **release** BIN
11. **else end** PRODUCTION-FIRING
12. **else if** I_k is DNT
13. **then while** BIN ownership doesn't change
14. **execute** tokens broadcast in BIN captured by SD
15. **if** I_k is still fireable **and** I_k has local actions
16. **then disable** local execution of any incoming token
17. **execute** local actions
18. **enable** local execution of incoming tokens

The test in step 7 is necessary because between the time the BIN was requested and the time its ownership is acquired, incoming tokens might have changed the status of the production selected to fire. If this occurs, the firing of the selected production is aborted. Steps 12-14 are executed for productions that are dependent on network transactions, as defined in section 3.1. If

such productions were to start firing while a remote processor is in the middle of a production execution, the intermingling of actions could result in non-serializable behavior. Notice that the BIN is released in step 10, before changes to local memory take place. To guarantee that no token is processed before the local changes are executed, buffering of tokens in BNB in step 15 is activated immediately upon releasing the BIN.

3.2.1 Detailed Processor Model

The processor architecture is detailed in Figure 3.2. The Instantiation Firing Engine (IFE) implements the outgoing interface with the Broadcasting Interconnection Network (BIN) and synchronizes internal activities. The IFE selects an instantiation to be fired among the ones stored in the Fireable Instantiation Memory (FIM). If the production selected to fire is global, the IFE places a request for ownership of the BIN[3]. Upon receiving BIN ownership, IFE waits until all outstanding tokens stored in BNB are processed. If the selected instantiation becomes unfireable due to such processing, IFE has to abandon it and select a new instantiation. Otherwise IFE broadcasts tokens with changes to the shared WMEs, releases the BIN, and executes the local actions.

The Snooping Directory (SD), along with the Broadcasting Network Buffer (BNB), implements the incoming network interface. The Snooping Directory is an associative memory that contains all WME types that belong to the antecedent sets of the productions assigned to the processing element[4]. BNB is used to store tokens broadcast on BIN and captured by SD during the local firing of a production, or during the execution of local actions of a global pro-

[3]The arbitration of the BIN is described in section 3.2.3.
[4]See the "Condition for Ownership", Axiom 3.1, Section 3.1.

Figure 3.2: Processing Element Model

duction. The tokens stored in BNB are processed as soon as the firing of the current production finishes. In the rare situation in which BNB is full, a halt signal is issued to freeze the activity on BIN. When the halt signal is reset, the activity in the bus resumes: the same processor that had BIN ownership continues to broadcast tokens as if nothing had happened.

Whether a WME change is originated locally or captured from BIN, it needs to be forwarded to the Rete Network and to the Fireable Instantiation Control (FIC). The Rete Network used in this architecture has β-memories. To avoid the high cost of waiting for the removal of a WME, which was pointed out

by Miranker [58], negated antecedents are stored in both β-memories and in the fireable instantiations produced for the conflict set. The presence of the negated conditions in this representation allows the quick removal of non-fireable instantiations when a new token is processed. There is a possibility that a WME change previously processed by FIC and not yet processed by Rete disables an instantiation freshly generated by Rete. To avoid a possibly non-serializable behavior, before adding a new instantiations to FIM, FIC checks it against the Pending Matching Memory (PMM), which stores all tokens still to be processed by Rete. The deletion of an instantiation from FIM is also performed by FIC. The operation of FIM, AFIM, PMM and FIC are explained in greater detail in section 3.2.2.

3.2.2 Conflict Set Management

The Fireable Instantiation Control (FIC) uses the Antecedents of Fireable Instantiation Memory (AFIM) to maintain a list of all enabled instantiations in the Fireable Instantiation Memory (FIM). AFIM and FIM are fully associative memories with capability to store don't cares in some of their cells. The fields in each line of FIM and AFIM are shown in Figure 3.3. FIC maintains an internal timer that is used to time stamp each instantiation added to FIM. Each line of AFIM stores either a WME that is the antecedent of a fireable instantiation, or an α-test that specifies an instantiation negated antecedent. Its fields are:

Presence - indicates whether the AFIM line is occupied. It is used to manage the space in the memory.

Negated - indicates whether this line stores a WME or a negated antecedent.

Type - stores the WME type.

Bindings - contains the values stored in each attribute-value pair of the WME. Notice that the name of the attribute doesn't need to be stored. Symbolic names are translated into integer values at compile time.

α-test - is used only for negated antecedents: specifies the α-test to be performed to verify a production antecedent.

Instantiation - indicates which fireable instantiations have this antecedent.

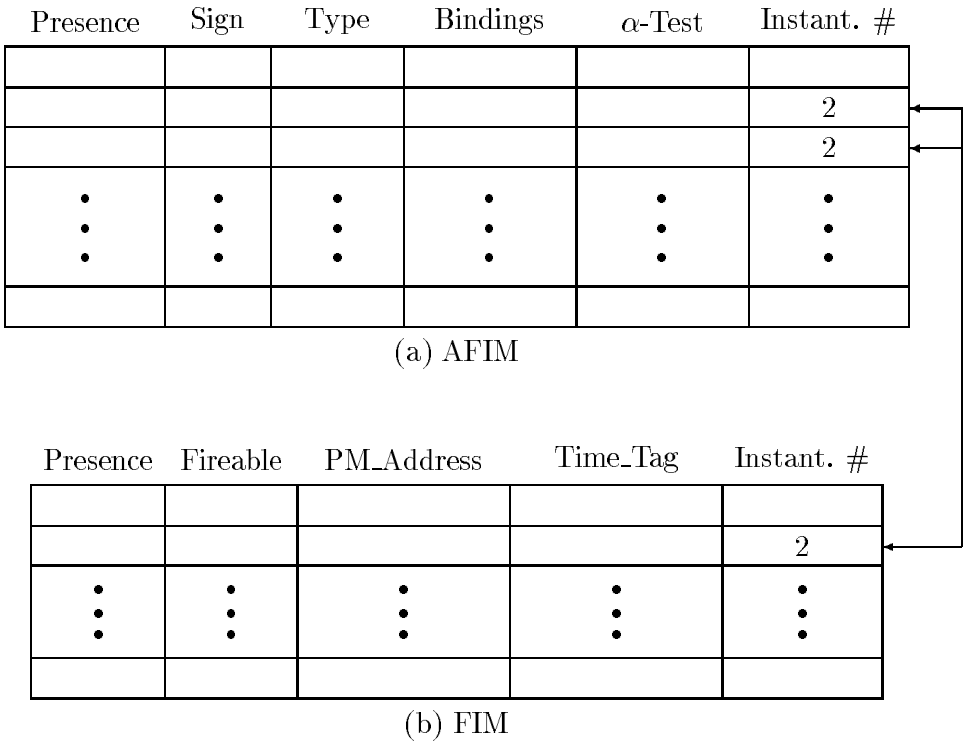

Figure 3.3: (a) Antecedents of Fireable Instantiation Memory; (b) Fireable Instantiation Memory.

Notice that because AFIM stores antecedents of fireable instantiations, most of the variables are bound, therefore the *bindings* field stores mostly constants.

For an easy handling of unbound variables, which match any value, the *bindings* field of AFIM is a ternary memory. Besides the values 0 and 1, it can also store a "don't care" value X. Such a memory might be implemented using two bits per cell, or using actual ternary logic in VLSI. One example of the latter is the Trit Memory developed by Wade [88]. One alternative to implement a non-bound value is to add a tag bit to *bindings* that indicates whether the value is bound or not. The advantage of this representation is that there is only one extra bit per word. Each line in FIM stores one fireable instantiation, with the following fields:

Presence - indicates whether the line is occupied;

Fireable - indicates whether the instantiation stored in the line is still fireable[5].

PM_Address - contains a pointer to the Production Memory indicating where the production actions are stored.

Time_Tag - record the time in which the instantiation became fireable. It is used to implement a *pseudo-recency criterion* to select an instantiation to be fired.

The third piece of memory managed by FIC is a fully associative memory called Pending Matching Memory (PMM). When a token is placed in the input nodes of the Rete Network, it is also stored in PMM. The token is removed from PMM when the Rete Network produces a signal indicating that all changes to the conflict set originated by that token have being processed. Upon receiving a new fireable instantiation from Rete, FIC associatively searches PMM[6]. If

[5] An instantiation is only removed from FIM after an incremental garbage collector removes the corresponding antecedents from AFIM.

[6] FIC has to perform an independent search for each antecedent of the new instantiation.

any line of PMM indicates the deletion (addition) of a WME that matches a non-negated (negated) condition of the instantiation, the new instantiation is ignored[7]. If no such line is found in PMM, FIC records the new instantiation in one line in FIM and stores each one of its antecedents in a separate line in AFIM. Figure 3.4 shows the organization of PMM with four fields:

Presence - indicates whether there is a WME stored in the line.

Sign - indicates whether this WME has been added to or deleted from the working memory.

Type - stores the type of WME.

Bindings - records the bindings of the WME.

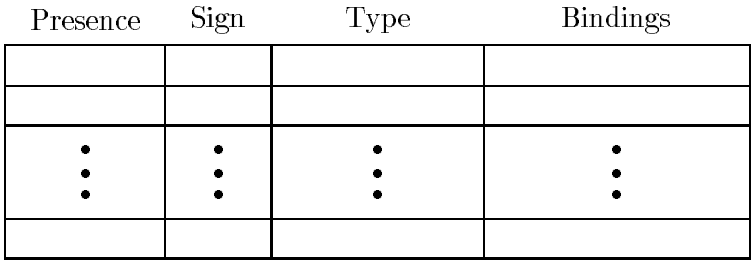

Figure 3.4: Pending Matching Memory

During the execution of a token, FIC performs three actions in parallel: send the token to the Rete Network input; add the token to PMM; and update FIM and AFIM. To update AFIM and FIM, first FIC executes an associative

[7]This instantiation must be ignored because the entry found in PMM indicates that a token received after the one that enabled the instantiation, which is not yet fully processed in Rete, will disable it.

search in AFIM for entries with the same WME present in the token, but with opposite sign. For each matching entry in AFIM, FIC marks the corresponding instantiation in FIM as unfireable. Finally FIC resets the presence bit for these entries in AFIM. This process leaves garbage[8] in FIM and AFIM. This garbage is all the non-fireable instantiations still present in FIM plus the antecedents of these instantiations in AFIM.

FIC has an *Incremental Garbage Collector* that searches FIM for an instantiation I_k that is non-fireable. FIC performs an associative search in AFIM and remove all antecedents of I_k, and then eliminates I_k from FIM. To guarantee the consistency of FIM and AFIM, the garbage collection is always performed as an atomic operation. For efficiency, the position in FIM in which the last garbage collection was executed is kept internally in FIC, and is used as the starting point of the next search. If FIM and AFIM are not full, garbage collection is performed at least once between two instantiation additions. Whenever FIM or AFIM are full, extra garbage collection is executed to free space. This solution trades memory space for speed: a WME that is tested by antecedents of many instantiations is stored many times in AFIM.

3.2.3 Broadcasting Interconnection Network Arbitration

Access arbitration in a broadcasting network is a well studied problem. In this machine we adopt the scheme used in the first prototype of the Alpha architecture by DEC [86]. During startup of a machine with n processors, each processor is assigned an arbitrary priority number from 0 to $n-1$. $n-1$ is the highest priority and 0 is the lowest. When a processor requests the network,

[8] "Garbage" is defined as a piece of data that still occupies space in memory, but will not be referenced again.

it uses its priority. The requester with highest priority is the winner and is granted access to the network. The winner has possession of the network as long as it needs to write all consequents of *one* production. After releasing the network, the winner sets its own priority to zero. All processors that had a priority number less than the winner increment their priority number by one, regardless of whether they made a request.

This scheme works as a round robin arbitration if all processors are requesting the network at the same time. If fewer processors are requesting the network, this mechanism creates the illusion that only these active processors are present in the machine.

In section 3.2 we establish that broadcast writes need to be kept in a buffer while a processor is firing local productions. When this buffer overflows, a *halt* signal is issued by the processor. This signal stalls all network broadcasting activities, giving time for the overloaded processor to consume its tokens and alleviate its buffer load. When the stall signal is removed, the network continues its activity without any change in the ownership. To avoid a great impact in the speed of the machine, the buffer must be sufficiently large to avoid frequent stalling of the network.

3.3 Correctness of the Processing Model

This section investigates whether the machine proposed in section 3.2 correctly executes a production system. The correctness criterion used is serializability [71], defined below.

Definition 3.15 (Serializability Criterion of Correctness) *The parallel execution of a collection of production instantiations $\mathcal{I}$ is correct iff there exists*

at least one serial execution of $\mathcal{I}$ that produces the same results as the parallel execution.

Theorem 3.1 *Giving the parallel machine model presented in this document, the definition of local DNT, local INT, and global productions, Axiom 3.1 is a necessary and sufficient condition of ownership to guarantee correct execution of a production system under the serializability criterion of correctness.*

Proof:

First we prove that axiom 3.1 is necessary. For the sake of contradiction, suppose that the ownership condition stated in axiom 3.1 is not satisfied. Assume that there is a production $R_n \in P_i$ and a WME W_k, such that $W_k \in A(R_n)$ and W_k is not stored in the local memory of P_i. Because reading operations are not allowed in the broadcasting network, P_i cannot perform the matching of R_n. Therefore a production system cannot be executed in such a machine. Thus, axiom 3.1 is necessary.

To prove that axiom 3.1 is sufficient, we must show that, in every possible circumstance, the results produced by this model could be obtained by a sequential execution of the productions. Therefore, we must analyze all situations in which parallel execution might occur and show that each one of them results in a serializable outcome. Because there is no parallel production firing within a processor, the following analysis is restricted to concurrent firing of productions allocated to distinct processors. Inter-processor parallelism occurs in two situations: among productions firing locally in distinct processors and between a production being broadcast over the in BIN and

one (or more) firing locally. All situations described below involve two productions being fired concurrently.

Situation 1: *Productions that have only local WMEs in its antecedents and consequents.*

The fact that all antecedents and consequents are local indicates that the productions being fired in parallel are completely independent of productions allocated to other processors, therefore the same results produced by the parallel firing could be obtained by any sequential firing of the same productions.

Situation 2: *A production $R_m \in P_j$ enables a production $R_n \in P_i$; R_m and R_n might have non-conflicting shared outputs; R_m does not disable R_n; R_n fires locally.*

Since R_n fires locally, all WMEs that are changed by both R_n and R_m are pseudo-local for P_i and shared for P_j. Because those are non-conflicting outputs and R_m enables R_n, parallelism occurs when R_n starts firing after being enabled by an action of R_m and before R_m finishes broadcasting changes to the network. The firing of R_n prevent the changes broadcast by R_m from being processed locally until R_n finishes. As long as the actions broadcast by R_m are queued and processed after R_n finishes, the result is the same as if R_n would have been fired after R_m finished. Thus, it is serializable.

Situation 3: *A production $R_m \in P_j$ disables a production $R_n \in P_i$; there is no enabling dependencies between R_m and R_n; R_m and R_n might have non-conflicting shared outputs; R_n fires locally.*

The only possibility for the parallel firing of R_m and R_n is for P_i to start firing R_n before P_j had broadcast any action that disables R_n. Even if P_j had broadcast some of the shared non-conflicting outputs when R_n starts firing, the effect is the same as firing R_n before R_m. Therefore, the result is serializable.

Situation 4: *A production $R_n \in P_i$ changes a pseudo-local WME W_k and a production $R_m \in P_j$ modifies W_k. R_n fires locally.*

In this situation, it is necessary to analyze three different cases:

Case 1: W_k is the only shared output between R_n and R_m.

Notice that the (possibly) conflicting WME W_k is exclusively stored in P_i[9]. Therefore if P_i disables the BIN before P_j broadcast changes to W_k, the result is the same of firing R_n before R_m. If P_i disables BIN after changes to W_k are broadcast, the result is equivalent to firing R_n after R_m. In both cases it is serializable.

Case 2: R_n and R_m have more than one shared output, but no more than one of them is conflicting.

The concern with multiple shared outputs is that the actions of the local and the global production might be intermingled. This would happen if P_i would inhibit actions from the network after P_j broadcast some but not all actions of R_m. Since R_m has only one action conflicting with R_n, the interruption of the

[9]Otherwise R_n could not fire locally.

remote firing will either take place before or after this conflicting action is broadcast. If the interruption occur before the conflicting action is executed in P_i, the result is equivalent to R_n firing before R_m. If it occurs after, the result is equivalent to R_n firing after R_m. In either case this situation results in a serializable behavior.

Case 3: R_n and R_m have more than one conflicting action.

In this case, if intermingled execution would be allowed, non-serializable behavior would result. However, according with condition (ii) of definition 3.13 this production is DNT and therefore cannot start firing until the network changes ownership, indicating that the global production either has finished or has not started. This ensures the necessary synchronization and results in serializable behavior.

Situation 5: *A production $R_n \in P_i$ is enabled and disabled by a production $R_m \in P_j$; R_n fires locally.*

In this situation, there would be a non-serializable behavior if production R_n would be allowed to fire after P_j had broadcast the action that enables R_n and before the action that disables R_n is broadcast. This situation does not occur because, according to condition (i) of definition 3.13, R_m is DNT: it only starts firing when the network changes ownership.

Situation 6: *A production $R_n \in P_i$ is enabled by a production $R_m \in P_j$; R_n has one output conflict with R_m; R_n and R_m may or*

may not have shared non-conflicting writes; and R_n fires locally.

Parallelism occurs if R_n starts firing in P_i after the action that enables R_n have been broadcast by P_j and before P_j finishes broadcasting R_m actions. If at that point the conflicting action has been already broadcast, the result will be equivalent to firing R_n before R_m. If the conflicting action has not been broadcast, the result is equivalent to R_m firing before R_n. Either way, the result is serializable.

Situation 7: *A production $R_n \in P_i$ is disabled by a production $R_m \in P_j$; R_n has one output conflict with R_m; R_n and R_m may or may not have shared non-conflicting writes; R_n fires locally.*

This situation could result in non-serializable behavior if R_n were to start firing after P_j broadcasts the conflicting action of R_m, and before the action that disables R_n is broadcast. However, this cannot occur because, according to condition (iii) of definition 3.13, R_n is DNT.

Situations 1 through 7 deal with possible dependencies involving two productions R_m and R_n allocated to distinct processors. The local firing of R_n in all situations indicates that its consequents change only local or pseudo-local WMEs. Table 3.1 helps to verify that every possible combination of dependencies among two productions in this situation have being analyzed. In this table a "-" indicates no dependencies, "1" indicates one dependency, "2+" indicates two or more dependencies, and "X" indicates any number of dependencies. Table 3.1 has five columns: "Enabling" column indicates the

number of actions in $C(R_m)$ that enable R_n; "Disabling" indicates the number of actions in $C(R_m)$ that disable R_n; "Non-Conflicting Write" indicate the number of non-conflicting shared actions between R_n and R_m; "Non-Conflicting Write" indicate the number of non-conflicting shared actions between R_n and R_m; and "Situation" indicates which of the situations analyzed in this proof covers each case. Every possible combination of dependencies between two productions is covered in table 3.1.

There is still the possibility that dependencies involving more than two productions create a situation in which the parallel model yields a non-serializable behavior. The only situation in which this might occur are in cycles of disablings, analyzed in situation 8.

Situation 8: *There is a cycle of disablings among productions allocated to distinct processors.*

First we analyze the special case in which the cycle is formed by two productions $R_n \in P_i$ and $R_m \in P_j$. According to definition 3.11, if there is a cycle of disabling between R_n and R_m, there exist two WMEs W_k and W_l such that $S_{C(R_m)}[W_k] \neq S_{A(R_n)}[W_k]$ and $S_{C(R_n)}[W_l] \neq S_{A(R_m)}[W_l]$. Therefore W_k is a shared WME for P_j, W_l is a shared WME for P_i, and neither R_m or R_n can fire locally. The acquisition of the broadcasting network works as a synchronizing element preventing R_n and R_m from firing in parallel. The same reasoning can be extended to disabling cycles with any number of productions.

Enabling	Disabling	Non-Conflicting Write	Conflicting Write	Situation
-	-	-	-	1
1	-	X	-	2
2+	-	X	-	2
-	1	X	-	3
-	2+	X	-	3
-	-	1	-	4, case 1
-	-	2+	-	4, case 2
-	-	X	1	4, case 2
X	X	X	2+	4, case 3
1	1	X	X	5
1	2+	X	X	5
2+	1	X	X	5
2+	2+	X	X	5
1	-	X	1	6
2+	-	X	1	6
X	1	X	1	7
X	2+	X	1	7

Table 3.1: Possible dependencies between R_n and R_m.

This concludes the proof. Since the results are serializable for any possible conflicting situation, we conclude that Axiom 3.1 is a sufficient condition of ownership and that the results produced by the model proposed are serializable.

□

Chapter 4

Benchmarking

A well known weakness of production system machine research is the lack of a comprehensive and broadly used set of benchmarks for evaluation of performance. In the process of searching for benchmarks to evaluate this novel architecture, we contacted many researchers with the same problem: a new idea to be evaluated in need of a suitable set of benchmark programs. Most of the benchmarks obtained were toy programs with a small number of productions in which the researcher can only change the size of the database. A benchmark in which the number of productions and the database size can be independently changed would allow researchers to study various aspects of new architectures. Section 4.1 presents a new benchmark that has such characteristics. It is a modification of the well known Traveling Salesperson Problem that we call a *Contemporaneous TSP* (CTSP). Another benchmark that we wrote is a solution to the "Confusion of Patents Problem". The following sections describe CTSP in detail and briefly present some other benchmarks used to test the architecture.

4.1 A Contemporaneous TSP

In this modified version of the TSP, the cities are grouped into "countries". The tour has to be constructed such that the salesperson enters each country only once. The location and borders of the countries must allow the construction of a tour observing this restriction. The problem is formally stated as follows:

> An instance of CTSP is represented by $(K, C, c, \mu_c, \sigma_c, O, d)$. $K = \{C_1, C_2, ..., C_n\}$ is a "continent" formed by "countries". Each country $C_i = \{c_{i,1}, c_{i,2}, ..., c_{i,m(i)}\}$ contains $m(i)$ "cities". The number of cities per country $m(i)$ is normally distributed with average μ_c and standard deviation σ_c. The ordering $O =< C_{\pi(1)}, C_{\pi(2)}, ..., C_{\pi(n)} >$ specifies the order in which the countries shall be visited. The function $d(c_i, c_j) \in Z^+$ specifies the distance between any two cities in the continent. The problem consists of finding an ordering of cities $< c_{i,\tau(1)}, c_{i,\tau(2)}, ..., c_{i,\tau(m(i))} >$ within each country C_i that minimizes the cost of the global tour:

$$\sum_{i=1}^{n} \sum_{j=1}^{m(i)-1} d(c_{i,\tau(j)}, c_{i,\tau(j+1)}) \quad + \quad \sum_{i=1}^{n-1} d(c_{i,\tau(m(i))}, c_{i+1,\tau(1)}) \; + \; d(c_{n,\tau(m(n))}, c_{1,\tau(1)}). \qquad (4.1)$$

This formulation of TSP is called "contemporaneous" because it reflects some aspects of modern day life. In the current global economy, travelpersons are likely to have greater needs than the traditional salesperson driving from town to town. Consider a music star in a worldwide tour carrying along a huge crew and sophisticated equipment: the singer will visit many different locations in each continent; the cost of flying back and forth between continents is much

higher than movements within a continent and depends on the locations of departure and arrival. Other situations involving sophisticated traveling requirements include the planning of airline routes and national political campaigns in large countries such as USA, Brazil and India. Applications in which data locality allows the creation of clusters include: insurance database management, banking industry, a national health care information network, and a national criminal offense information network[1].

4.1.1 A Production System Solution for CTSP

The formulation presented above for the CTSP is generic enough to allow its application in many fields: there is no restriction in what the words continent, country, city, and distance might represent. To facilitate the construction of a Production System solution that is useful for testing new PS architectures, we used a simpler version of CTSP with the following restrictions:

- The problem is symmetric, i.e., $d(c_{k,i}, c_{l,j}) = d(c_{l,j}, c_{k,i})$ for any i, j, k, and l.
- A continent is a two-dimensional Euclidian space.
- A country is a contiguous, rectangular shape within this space.
- The number of cities in each country follows a normal distribution with average μ_c and standard deviation σ_c.
- The city locations are uniformly distributed within each country.

[1] In the 1994 "Brady Bill", Congress mandated the construction of such a network for background verification for the purchase of fire weapons.

- There is a common boundary between two countries that are consecutive in the ordering O.

Our PS solution for CTSP has a set of productions for each country and a set of productions for each country boundary. The data set is constructed in such a way that the distances among cities located within each country are stored in WMEs with different types. Given a country C_i, the country that precedes C_i in the order O is denominated $P(C_i)$; the country that succeeds C_i in the order O is denominated $S(C_i)$. It is not necessary to store in the data base the distance between every two cities in the continent. For a city $c_{i,j}$ in a country C_i, the only relevant distances are the distance to the cities within C_i, to the cities in $P(C_i)$, and to the cities in $S(C_i)$. The following list illustrates WMEs typically used in our solution to CTSP:

```
(GERMANY_city ^name GERMANY_01 ^status not_in_trip)
(FRANCE_city ^name FRANCE_10 ^status in_trip)
(GERMANY_dist ^from GERMANY_04 ^to GERMANY_07 ^value 135)
(FRANCE_GERMANY_dist ^from FRANCE_14 ^to GERMANY_03 ^value 357)
(GERMANY_POLAND_dist ^from GERMANY_01 ^to POLAND_05 ^value 55)
```

Our solution has seventeen *local* productions per country and twelve productions per country boundary. This organization allows the researcher to vary the number of productions by creating continents with different number of countries. The size of the data base is determined by the number of countries and the average number of cities per country. The variance between the amount of data processed by each cluster of production is given by σ_c.

The heuristic used in the PS solution of the problem involves the computation of two extra locations for each country C_i: the geometric center of the

borders with $P(C_i)$ and with $S(C_i)$. Because we impose the restriction that countries have rectangular shapes in a two-dimensional Euclidian space, the border between two subsequent countries in the tour is always a segment of a straight line. The *border center* $b(C_i, C_j)$ between countries C_i and C_j is the center of the line segment that forms the boundary. The heuristic used to construct the internal tour in a country C_i is described below:

- The first city $c_{i,k}$ in the internal tour of a country C_i is the city with minimum distance $d(b(C_i, P(C_i)), c_{i,k})$.
- While the internal tour of country C_i is not complete, select a city $c_{i,}l \in C_i$ such that $d(c_{i,k}, c_{i,l}) - d(c_{i,l}, b(C_i, S(C_i)))$ is minimum, where $c_{i,k}$ is the latest city inserted in the tour.
- Whenever the internal tours of two adjacent countries C_i and C_j are completed, the last city visited in C_i is connected to the first city visited in C_j.
- Whenever there is a segment of tour formed by four cities (c_i, c_j, c_k, c_l) such that $d(c_i, c_j) + d(c_k, c_l) > d(c_i, c_k) + d(c_j, c_l)$, change this segment of tour to (c_i, c_k, c_j, c_l)[2].

This rationale of the heuristic is to add to the internal tour the cities that are close to the latest city included in the tour and far from the border in which the internal tour shall end. There is a limited local optimization of the constructed tour. We developed a C program that allows researchers to specify continent maps and to experiment with different numbers of countries, μ_c, and σ_c.

[2]The first subscript in the notation $c_{i,j}$ is ommited here because these local optimization might occur either within a country or accross country's borders.

Two production system solutions were constructed for CTSP. In the first one, identified as `tsp` in Table 4.1, a single set of productions performs the optimization in all country borders. In the second solution, identified as `tsp2` in Table 4.1, an specilized set of productions is used in the optimization of each country border. Table 4.1 presents static measures for instantiations of CTSP considering problems with C countries, with each country having an average of μ_c cities.

Measure	`tsp`	`tsp2`
# of productions	$20\,C + 1$	$30\,C + 1$
# WME types	$8\,C + 8$	$15\,C + 1$
# WMEs in initial database	$C\,(2\,\mu_c^2 + 2\,\mu_c + 3)$	$C\,(2\,\mu_c^2 + 2\,\mu_c + 3)$

Table 4.1: Static measures for the CTSP benchmark according to C and μ_c

Figure 4.1 plots the number of WMEs in the knowledge base during the execution of the benchmark `moun2`, which is a CTSP with ten states and $\mu_c = 15$. The number of WMEs grows steadily from its initial value as the tour is constructed and the distance among cities are not removed. However, there is not much variation overall in the amount of data stored in the knowledge base during the execution of the benchmark.

This simplified CTSP offers many advantages for production system benchmarking: the number of productions in the program can be varied by changing the number of countries; the ratio of global to local data is controlled by the average number of cities in each country; the balance in the size of local data clusters is specified by σ_c; and the specification of the continent "map" is very simple making it easy for a researcher to generate new instantiations of the benchmark. Observe from table 4.1 that the relation between the number of WME

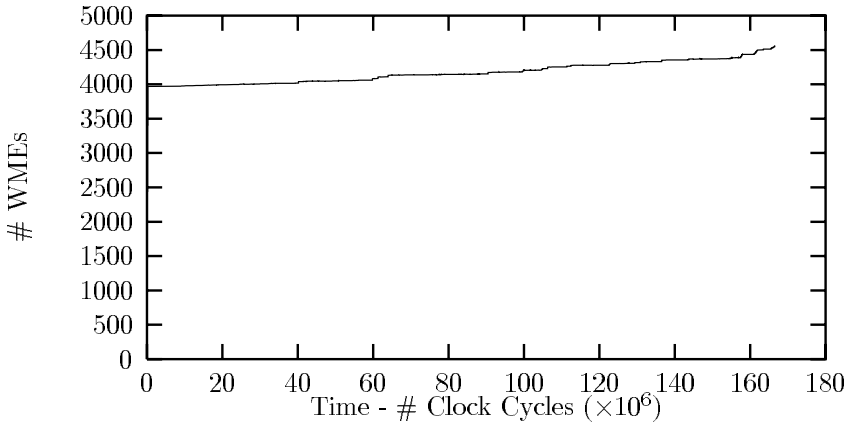

Figure 4.1: Variation in the knowledge base size during the execution of `moun2`.

types and the number of productions is constant (≈ 2.5). The CTSP benchmarking facility is available through anonymous ftp to: `pine.ece.utexas.edu` in `/a/pine/home/pine/ftp/pub/parprosys`. In the measurements presented in section 6.1, instances of the CSTP appear as `south`, `south2`, `moun` and `moun2`.

4.2 Confusion of Patents Problem

We constructed a solution for the formulation of the *Confusion of Patents Problem* presented in [19, 40] The problem presents five patents, five inventors, five cities, and ten constraints. Using these constraints we must decide who invented what and where. In our solution, all 125 possible combinations and 10 constraints are present in the initial database; 67 productions use the constraints to eliminate combinations that are not possible; 19 productions select the right combinations and print the solution. 1

Because this solution has only four different types of WMEs, most of the pro-

ductions either change or test the same kinds of WME. As a consequence, productions have strong interdependency, resulting in a production system poorly suited for clustering. Even in a machine with a moderate number of processors, most of the actions need to be broadcast on the network. The main source of parallelism is the concurrent execution of different portions of the Rete network. Performance measures to this solution of the confusion of patents problem are reported under the name `patents`.

4.3 The Hotel Operation Problem

Originally written by Steve Kuo at the University of Southern California, `hotel` is a production system that models the operation of a hotel. It is a relatively large and varied production system (80 productions, 65 WME types) with 17 non-exclusive contexts. Because each production in `hotel` is related with the activities that actually take place in a hotel, the amount of speedup obtained depends on the balance of work among each one of these activities. For example, if a hotel is specified with a large number of tables in the restaurant and very few rooms, the productions that take care of the restaurant tables will have a much larger load than the productions that cleanup the rooms. This work unbalance is transferred to parallel architectures that partition the program at the production level.

4.4 The Game of Life

This is an implementation for Conway's game of life, as constructed by Anurag Acharya. After our modifications, `life` has forty productions. Twenty five of these productions are in the context that computes the value of each

cell for the next generation and potentially can be fired in parallel. The other fifteen productions are used for sequencing and printing and can be only slightly accelerated by Rete network parallelism.

4.5 The Line Labeling Problem

Different versions of the line labeling problem (Waltz and Toru-Waltz) have being used for performance evaluation [49, 51, 63, 70]. Our version was originally written by Toru Ishida (Columbia Univ.), and successively modified by Dan Neiman (Univ. of Massachusetts), Anurag Acharya (Carnegie-Mellon Univ.) and José Amaral (Univ. of Texas). The current version has two non-overlapping stages of execution, each one with four productions. Because the system is partitioned at the production level, the amount of parallelism is limited to four fold. Such a low limit in speedup occurs because this is a simple "toy" problem with only ten productions, not adequate for the architecture proposed. The line labeling problem is identified as `waltz2` in our set of benchmark.

4.6 Benchmark Static Measures

Table 4.2 shows static measures — number of productions, number of distinct WME types, average number of antecedents per production, average number of consequents per productions, and number of WMEs[3] — for the benchmarks used to estimate performance in the multiple functional unit Rete network. `south` and `south2` are CTSPs with four countries and ten cities per

[3]The number of WMEs changes during program execution. The number reported in Table 4.2 is the number of WMEs in the initial knowledge base.

Bench.	# Prod	Ant./prod	Cons./prod	# WME types	# WMEs
life	40	6.1	1.3	5	104
hotel	80	4.1	2.0	62	484
patents	86	5.2	1.2	4	136
south	91	4.7	2.8	40	774
south2	121	4.7	2.7	61	774
moun	211	4.7	2.8	88	3970
moun2	301	4.7	2.7	151	3970
waltz2	10	2.7	8.0	7	60

Table 4.2: Static measures for benchmarks used.

country; `moun` and `moun2` are CTSPs with ten countries and 15 cities per country; `life`, `patents`, `waltz2`, and `hotel` are the benchmarks discussed in sections 4.2 to 4.5.

Chapter 5

Production Partitioning Algorithms

The problem of partitioning a Production System into disjoint sets of productions which are then mapped onto distinct processors has been studied by a number of researchers. Most partitioning algorithms are designed with the goal of minimizing or reducing enabling, disabling, and output dependencies among productions allocated to different processors [70]. Oflazer formulates partitioning as a minimization problem and concludes that the best suited architecture for Production Systems has a small number of powerful processors [64]. Oflazer also indicates that a limited amount of improvement in Production System machine speed can be obtained by an adequate assignment of productions to processors. Moldovan presents a detailed description of production dependencies and expresses the potential parallelism in a "parallelism matrix" and the cost of communication among productions in a "communication matrix" [59]. Xu and Hwang use a similar scheme with matrices of cost to construct a simulated annealing optimization of the production partition problem [92].

Although certain basic principles are maintained in all partitioning schemes,

partition algorithms are tailored to specific architectures. We are concerned with two kinds of relationships among productions: productions that share antecedents, and productions that have conflicting actions. In the architecture described in chapter 3, assigning productions with common antecedents to the same processor reduces memory duplication, while assigning productions with conflicting actions to the same processor prevents traffic in the bus.

Previous partition algorithms were greatly influenced by enabling and disabling dependencies among productions [59, 64, 92]. Our experience with production systems indicates that grouping productions with common antecedents is much more effective to reduce the communication cost. Moreover, in the production system programs that we examined, a production seldom creates a WME that was not tested by its antecedents. Therefore, productions that have more common antecedents are also most likely to have a greater number of enabling and disabling dependencies among them. Thus, our partition algorithm does not directly consider these dependencies, but only shared antecedents and conflicting outputs.

5.1 Production Relationship Graph

To represent the relationships among productions we define an undirected, fully connected graph PRG = (R, E) called *Production Relationship Graph*. Each vertex in R represents one of the productions in the system, and each weighted edge in E is a combined measure of the relationships of the two productions represented by its vertices. PRG has a weight function $w : E \rightarrow Z^+$, defined by equation 5.1.

$$w(E_{ij}) = w(E_{ji}) = (1-\delta_{ij}) \sum_{l=0}^{n-1} \sum_{k=0}^{m-1} \psi_{li,kj} + (1-\delta_{ij}) \sum_{l=0}^{p-1} \sum_{k=0}^{q-1} \gamma_{li,kj}, \quad (5.1)$$

where n and m are the number of antecedents and p and q are the number of consequents in productions R_i and R_j, respectively, δ_{ij} is 1 if $i = j$ and 0 otherwise, and

$$\psi_{li,kj} = \begin{cases} 1 & \text{if antecedents } W_l \text{ of } R_i \text{ and } W_k \text{ of } R_j \text{ are of the same type.} \\ 0 & \text{otherwise} \end{cases}$$

$$\gamma_{li,kj} = \begin{cases} 1 & \text{if consequent } W_l \text{ of } R_i \text{ conflicts with } W_k \text{ of } R_j \\ 0 & \text{otherwise} \end{cases}$$

The first term of equation 5.1 computes the number of common antecedents between productions R_i and R_j. The second term computes the number of output conflicts between the productions. If two productions connected in PRG by an arc with heavy weight are allocated to distinct processors, the communication cost increases because these productions will most likely be remote or dependent on network transactions (DNT). Therefore to reduce traffic in the bus, it is necessary to partition the productions among processors by cutting edges with lighter weights in PRG. A cut in a graph is defined as follows [13].

Definition 5.1 *A partition of R in the graph $PRG = (R, E)$ is a* **cut** $(S, R-S)$. *An edge $(u, v) \in E$* crosses *the cut $(S, R-S)$ if and only if one of its endpoints is in S and the other is in $R-S$.*

5.2 The Minimum Cut Problem

The optimal partitioning of productions into disjoint sets can be modeled as a minimum cut problem. The minimum cut problem is the problem of generating a cut in a full connected graph in such a way that the sum of the weights of the edges that cross the cut is minimum. When there is a limit for the number of vertices in each set, the problem is called *minimum cut into bounded sets* [21]. This problem was proven to be NP-Complete by means of reducing the *simple max cut* problem, by Gary, Johnson and Stockmeyer in 1976 [22].

In this chapter we present and evaluate the performance of four distinct polynomial time algorithms that produce sub-optimal partitions of a production set. Algorithms 1 and 3, based only on the compile-time information provided in the PRG, partition the production into almost equal size subsets. Their goal is to minimize the number of shared antecedents and conflicting outputs among productions in different subsets. Algorithms 5 and 6[1] use run-time information obtained from measurements in previous executions of the same program to balance the workload on different processors, even though this might result in subsets with different sizes.

5.3 Algorithm 1

This algorithm is inspired by the minimum cut problem. It recursively divides the production set in half until the number of subsets is identical to the number of processors in the machine. The algorithm constructs the partition

[1] The algorithm numbers were assigned as they were designed. Because we are not presenting some of the algorithms here, some numbers are missing.

in such a way that the subset with the most productions has at most one more production then the subset with the least productions. The parameters for the algorithm PARTITION-1(R, E, w, n, M, C) are the set of vertices P, the set of edges E, the weight function w, the number of partitions needed n, the number of productions to be allocated to one subset in the next cut M, and the set of subsets C that stores the partition. Initially, R includes all the productions, C is empty, and n is equal the number of processors in the parallel machine. A global variable, $m = \lfloor |R|/n \rfloor$ denotes the minimum number of productions per subset in the final partition.

```
PARTITION-1(R, E, w, n, M, C)
1   if n = 2
2     then S ← TWO-PARTITIONS(R, E, w, ⌊|R|/2⌋)
4          C ← C ∪ S
5          C ← C ∪ (R−S)
6          return
7   if n = 3
8     then S ← TWO-PARTITIONS(R, E, w, ⌊|R|/3⌋)
9          C ← C ∪ S
10         PARTITION-1(R−S, E, w, 2, m, C)
11         return
12  if ⌈M/2⌉ < m⌈n/2⌉
13    then A ← M − m⌈n/2⌉
14    else  A ← ⌊M/2⌋
15  S ← TWO-PARTITIONS(R, E, w, A)
16  PARTITION-1(S, E, w, ⌈n/2⌉, M−A, C)
17  PARTITION-1(R−S, E, w, ⌊n/2⌋, A, C)
```

TWO-PARTITIONS uses the function *cut cost* $cc : R \rightarrow \mathrm{R}$ to decide which production it selects to transfer to a new subset.

$$cc(R_i) \quad = \quad \frac{\sum_{j=0}^{r-1} w(E_{ij})\,\phi_{ij}}{1 + \sum_{j=0}^{r-1} w(E_{ij})\,(1 - \beta_{ij})}, \tag{5.2}$$

where r is the number of productions being considered, i.e., $r = |R|$, $w(E_{ij})$ was defined in eq. 5.1, and

$$\phi_{ij} = \begin{cases} 1 & \text{if } [R_i \in S \textit{ and } R_j \in (R-S)] \textit{ or } [R_i \in (R-S) \textit{ and } R_j \in S] \\ 0 & \text{otherwise.} \end{cases}$$

The function $cc(R_i)$ determines the quotient between the sum of the edges that are connected to the vertex of R_i and that cross the cut $(S, R-S)$, and the sum of the edges that are connected to the vertex of R_i and do not cross the cut $(S, R-S)$. A high value for $cc(R_i)$ indicates that the most strong connections of the production R_i are crossing the cut, and therefore R_i should be transferred to the other set.

TWO-PARTITIONS(R, E, w, M)
1 $S \leftarrow \emptyset$
2 $S \leftarrow S \cup \{R_i \,/\, W(R_i)$ is maximum $\}$
3 **for** $i \rightarrow 1$ **to** $M-1$
4 **do** $S \leftarrow S \cup \{R_i \,/\, R_i \in (R - S)$ and $cc(R_i)$ is maximum $\}$
5 **return** S

The parameter M in TWO-PARTITIONS indicates the number of productions to be transferred to the new set S. The first production to be moved to S is the one that has the maximum *cumulative weight*. The production with maximum cumulative weight has strong connections in PRG with other productions and will attract those productions to the new set. The cumulative weight of a vertex is measured by the function $W : R \rightarrow \mathrm{Z}$ by equation 5.3.

$$W(R_i) = \sum_{j=0}^{N-1} w(E_{ij}), \tag{5.3}$$

where N is the total number of productions in the production system program.

For the analysis of time complexity of the algorithms presented in this chapter, the *asymptotic upper bound* O-notation is used. This notation was introduced as early as 1892, and its use is advocated by Knuth [46], and Cormen *et al.* [13].

PRG can be built in $O(N\,(a^2+c^2))$ time complexity, where a is the maximum number of antecedents, and c is the maximum number of consequents in any production. In step 2 of TWO-PARTITIONS, it is necessary to compute the cumulative weight $W(R_i)$ of every production, which takes $O(N^2)$ time. The selection of the maximum can be done as part of the computation, without additional complexity; therefore, step 2 has complexity $O(N^2)$. For step 3, the values of $cc(R_i)$ are stored in a vector that is initialized with zeros (S is empty in the beginning). When a vertex is moved to S, this vector can be updated in $O(N)$ time and the new maximum can be selected also in $O(N)$ time. Step 3 has to execute up to $N/2$ times, resulting in a time complexity of $O(N^2)$ for step 3. Therefore, considering the construction of PRG, the complexity of TWO-PARTITIONS is $O(N\,(a^2 + c^2 + N))$. Assuming that $N \geq (a^2 + c^2)$,

this can be simplified to $O(N^2)$.

The running time $T(N,n)$ of PARTITION-1 is described by the recurrence

$$T(N,n) = \begin{cases} O(N^2) & \text{if } n = 2, \\ T(\lceil 2N/3 \rceil, 2) + O(N^2) & \text{if } n = 3, \\ T(\lfloor N/2 \rfloor, \lfloor n/2 \rfloor) + T(\lceil N/2 \rceil, \lceil n/2 \rceil) + N^2 & \text{if } n > 3, \end{cases}$$

where N is the number of productions to be partitioned, and n is the number of partitions to be produced. After i iterations, the recurrence becomes

$$T(N,n) = 2^i \, T\left(\frac{N}{2^i}, \frac{n}{2^i}\right) + N^2 \sum_{j=0}^{i-1} \frac{1}{2^j}$$

Replacing i by $\log n$ (the number of times PARTITION-1 actually iterates) and computing the summation, we obtain

$$T(N,n) = n \, T\left(\frac{N}{n}, 1\right) + 2 \, N^2 \left(1 - \frac{1}{n}\right)$$

that results in a time complexity of $O(n + N^2)$. With the reasonable assumption that the number of productions is much larger than the number of processors in the machine, the time complexity of PARTITION-1 becames $O(N^2)$.

5.4 Algorithm 3

One drawback of algorithm 1 is that once a cut is made and two productions are allocated to different subsets, there is no possibility of placing them into the same subset later on. In other words, after the first cut, the optimization is local to a subset of productions. To improve this aspect, we designed a recurrent algorithm that was assigned the number 3. *Algorithm 3* uses a *fitness* function

$F : R \times S \rightarrow \mathrm{Z}$ to choose the next production to be assigned.

$$F(R_i, S_k) \;=\; \sum_{j=0}^{N-1} w(E_{ij})\, \eta_{jk}, \tag{5.4}$$

where S_k is a subset of productions assigned to the k^{th} processor. Initially all productions are in R and all subsets S_k are empty, and

$$\eta_{jk} = \begin{cases} 2 & \text{if } R_j \in S_k \\ 1 & \text{if } R_j \in R \\ -1 & \text{if } R_j \in S_m \neq S_k \end{cases}$$

The value of the fitness function indicates how the production represented by the vertex R_i fits in the subset S_k. $F(R_i, S_k)$ computes a weighted sum of the connections between vertex R_i and all other vertices in PRG. A strong connection with a vertex that has been assigned to a set other than S_k reduces the fitness of R_i to S_k, while a strong connection with a vertex already in S_k increases the fitness. A strong connection with a vertex that has not been assigned to any subset has an intermediate value because S_k may be able to attract both vertices.

The arguments for algorithm 3 are the set of productions R, the arcs in PRG E, the weight function w, the fitness function F, and the number of processors in the machine n.

Algorithm 3 starts by placing one production in each subset, and then proceeds to add one more production to each subset until all productions are assigned. The efficiency of the algorithm is highly dependent on the initial placement. A pathological set of weights in PRG might cause this algorithm to produce a poor partition.

PARTITION-3(R, E, w, F, n)
1 **for** $k \leftarrow 0$ **to** $n-1$
2 **do** $S_k \leftarrow \{R_j$ / $F(R_j, S_k)$ is maximum $\}$
3 $R \leftarrow R - \{R_j\}$
4 **for** $k \leftarrow 0$ **to** $R \equiv \emptyset$ **step** 1 **mod** n
5 **do** $S_k \leftarrow S_k \cup \{R_i$ / $R_i \in R$ and $F(R_i, S_k)$ is maximum $\}$
5 $R \leftarrow R - \{R_i\}$

Initially, all productions are in R, and all subsets S_k are empty. Therefore the initial fitness is determined only by R_i and is identical to $W(R_i)$ defined by equation 5.1. The computation of $W(R_i)$ for all productions takes $O(N^2)$ time. The fitness is stored in an array whose initialization takes $O(nN)$ time. Where n is the number of processors in the architecture. Whenever a production is removed from R and placed in a subset S_i, only a constant number of positions in this array need to be modified and each modification can be done in time $O(1)$. If the maximum of a previous iteration is also stored, the new maximum can be found in $O(1)$ time. The for loop executes $(N - n)$ times. Therefore the time complexity is $O(N^2 + nN + N - n) = O((n + N)N)$. Assuming that $N \gg n$, we obtain a time complexity of $O(N^2)$ for algorithm 3.

5.5 Algorithm 5

Initial studies with a parallel architecture simulator showed that the main factor limiting further reduction in execution time is the time spent in the matching phase in the Rete Network. Consequently, we designed an algorithm that balances the load in each processor considering the firing frequency of each

production. The objective is to obtain an equal amount of matching assigned to each processor. We are assuming that productions fired more frequently also produce more activity in the matching engine. The amount of work assigned to each subset S_i of the partition is computed by the *work load* function $\mathcal{L} : S \rightarrow \mathrm{Z}$.

$$\mathcal{L}(S_i) \;=\; \sum_{j=0}^{N-1} A(R_j)\,\mathcal{F}(R_j)\,\gamma_{ij}, \tag{5.5}$$

where N is the total number of productions, $A(R_j)$ is the number of antecedents in production R_j, and $\mathcal{F}(R_j)$ is the firing frequency of production R_j measured in a previous execution of the same program, and

$$\gamma_{ij} \;=\; \begin{cases} 1 & \text{if } R_j \text{ is assigned to } S_i \\ 0 & \text{otherwise.} \end{cases} \tag{5.6}$$

(5.7)

PARTITION-5$(R, E, w, F, n, \mathcal{L})$
1 **for** $k \leftarrow 1$ **to** n
2 **do** $S_k \leftarrow \{R_i \;/\; F(R_i, S_k) = \max_j F(R_i, S_k)\}$
3 $R \leftarrow R - \{R_i\}$
4 **while** $R \neq \emptyset$
5 **do** $S_k \leftarrow S_k \cup \{R_i \;/\; R_i \in R$ and $\mathcal{L}(S_k) = \min_j \mathcal{L}(S_j)$
and $F(R_i, S_k) = \max_l F(R_l, S_k)\}$
6 $R \leftarrow R - \{R_i\}$

Algorithm 5 is similar to algorithm 3, except that instead of assigning one production to each processor regardless of their work load, a new production

is assigned to the processor with minimum work load. In the end the number of productions assigned to each processor might be different, but the work load will be evenly distributed among the processors.

Like in algorithm 3, an array with the initial values of the fitness can be constructed in time $O(N^2+nN)$. The values for the work load can also be kept in an array whose initialization takes time $O(n)$. Both the fitness array and the work load array can be updated in time $O(1)$ after each assignment. The while loop is repeated $N-n$ times. Therefore algorithm 5 has the same time complexity as algorithm 3: $O(N^2)$.

5.6 Algorithm 6

A more detailed analysis of matching in the Rete Network reveals that most of the matching time in consumed in the execution of tests in β-nodes. Our simulator allows for the measurement of the number of β-tests performed in the antecedents. Algorithm 6 uses this measurement to estimate the work load associated with each production. We define the function $B : S \rightarrow \mathrm{Z}^+$, which computes the number of β-tests that are expected to be performed by the productions placed in set S_i.

$$B(S_i) \;=\; \sum_{j}^{k} \mathcal{B}(R_j)\,\gamma_{ij}, \tag{5.8}$$

where k is the number of productions in set S_i, γ_{ij} has tha same definition as for algorithm 5, and $\mathcal{B}(R_j)$ is the number of beta tests performed for production R_j. $\beta(R_j)$ is measured in previous runs of the same production system.

The strategy used in algorithm 6 consists of selecting the processor with the

least number of beta tests, and then finding the production best fitted for this processor. Productions strongly related to other productions in PRG are the first ones to be assigned to processors. The algorithm ends when there are no more productions in P.

PARTITION-6(R, E, w, n, F, B)
1 **while** $S \neq \emptyset$
2 **do** $S_k \leftarrow S_k \cup \{ R_i \ / R_i \in R$ and
$B(S_k) = \min_j B(S_j)$ and
$F(R_i, S_k) = \max_l F(R_l, S_k)\}$
3 $S \leftarrow S - \{R_i\}$

The values for the function $B(S_k)$ can be stored in an array which initialization takes $O(n)$ time. The initialization of an array for the fitness function $F(R_i, S_k)$ takes $O(nN + N^2)$ time. If the minimum and maximum are stored, their values can be updated at every iteration of the loop in time $O(1)$. There is a constant number of positions in the arrays F and B to be updated at each interaction and their updating takes time $O(1)$. The while loop is executed in N times. Therefore the time complexity of algorithm 6 is also $O(N^2)$.

5.7 Simulation Results

Figure 5.2 shows speedup curves for the benchmarks `south2` and `patents` for machines with one up to twenty processors. Observe that algorithms 5 and 6, which use dynamic information about the amount of processing due to each production in previous run, are generally superior to algorithms 1 and 3 that

do not use such information. This superiority tends to appear more clearly in machines with a larger number of processors.

As Oflazer indicated earlier, the gain in speedup due to choosing a good partition algorithm is limited [64]. However, this gain is obtained without any extra hardware cost. Furthermore, all algorithms presented in this chapter have the same asymptotic time complexity and are straightforward to understand and implement. Therefore we recommend the use of algorithm 6 for the architecture model presented in Chapter 3.

The speed yielded by different partition algorithms is closely related to the performance of individual components of the machine. When we consider a multiple functional Rete Network in Chapter 7, the amount of time spent in matching will be reduced, and so will the importance of balancing the matching load. Consequently, the advantage of algorithms 5 or 6 over algorithms 1 or 3 will be diminished.

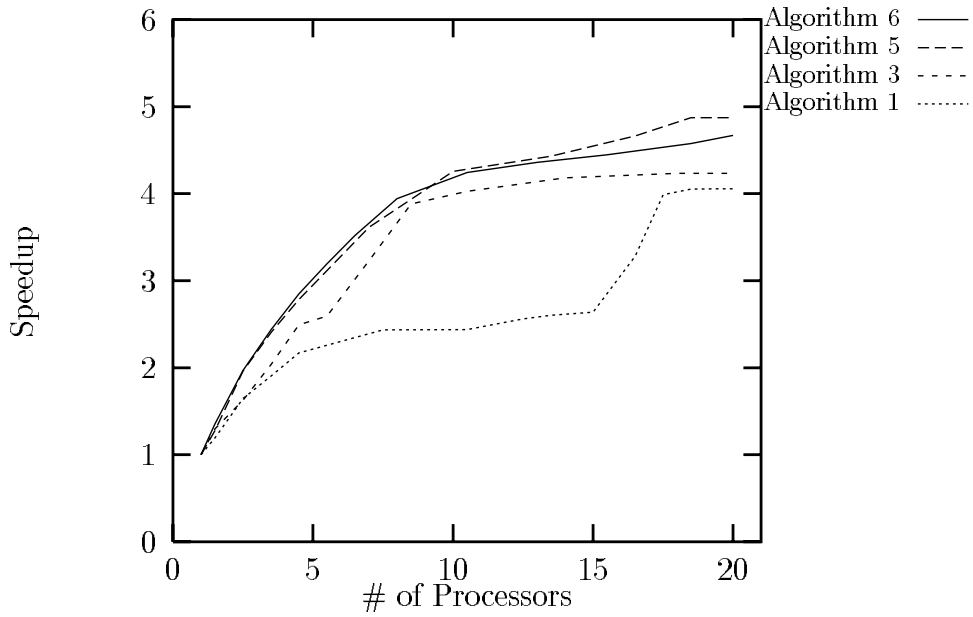

Figure 5.1: Partition Algorithm Speedup Curves for `south2`

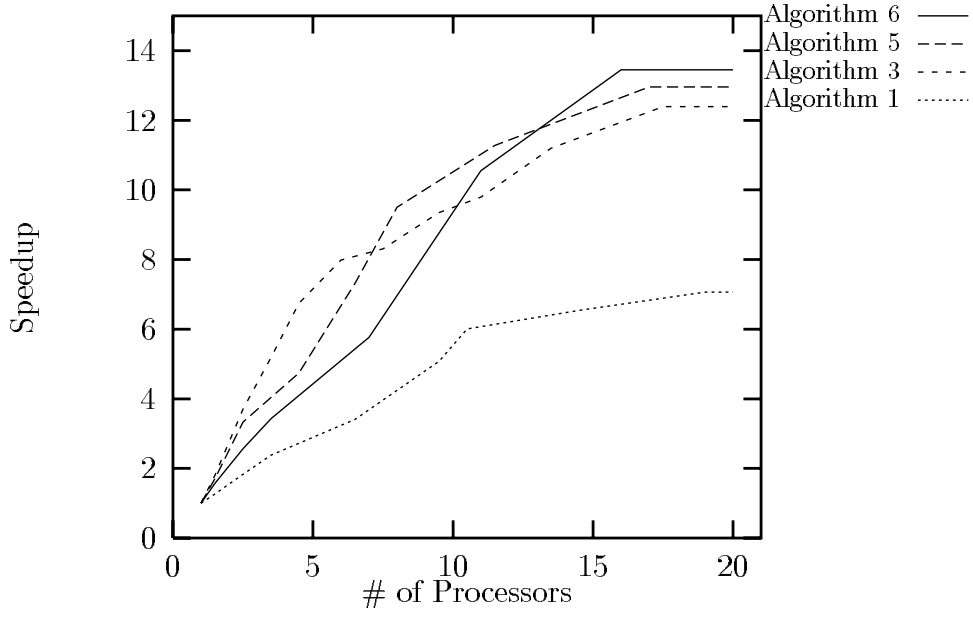

Figure 5.2: Partition Algorithm Speedup Curves for `patents`

Chapter 6

Performance Evaluation through System Simulation

> *"Although, on the surface, rule-based systems appear to have a lot of parallelism, in practice the true speed-up achievable from parallelism is quite limited, less than tenfold."*
> (Anoop Gupta *et al.*, 1989) [29].

To evaluate the performance of the architecture presented in Chapter 3, we developed an event driven simulator. The input to the simulator consists of production system programs written in OPS5 syntax. These programs are expected to be correct under the serializability criterion; the programmer must not rely on any knowledge about conflict set resolution strategy to guarantee correctness. For the syntax and lexical analysis, the tools *yyacc* and *yylex* were

used[1] [4].

The simulator performs the dependency analysis described in section 3.1 and implements the partitioning of productions according to algorithm 6 presented in Chapter 5. The representation of productions is augmented with the specification of its type — local DNT, local INT, or global; of the type of each one of the antecedents — local, pseudo-local, or shared; and the type of each one of the consequents — local or shared. This information is used by the IFE to decide the actions to be taken upon firing each production. After finishing the partitioning, the simulator creates the data structures necessary to represent each processor and the broadcasting network, and sets up the initial states of the Rete Networks using the initial database.

The simulator implements $N + 1$ independent processing units, namely the N processors plus the network controller. In the simulated architecture, many events might occur at the same time in different processors or even within a processor. For example, the matching in the Rete Network is completely concurrent with the snooping of the broadcasting network by the Snooping Directory and with the selection of an instantiation by the Firing Engine. Another situation is the broadcasting of a token in the network: the Snooping Directory of every processor needs to perform, at the same time, a search to verify if the token needs to be captured for local processing. To guarantee a timely and orderly execution of events in a sequential computer, the simulator has a main routine that does not reflect any specific structure in the machine but is necessary to control the sequencing of events. This main routine keeps an *event heap* that contains the earliest event in each unit. Each unit has a local event heap with

[1]The front-end conversion of the OPS5 syntax into internal data structure was built by Anurag Acharya at Carnegie Mellon University for PPL [1, 3].

all the events scheduled for that unit. The main routine selects the next event (the earliest in time), and calls the routine corresponding to that unit. In the process of executing the next event, this unit might generate one or more new events that are inserted in the local heap. Upon finishing the event execution, the unit returns to the main function the time of the new next event for that unit. The simulation terminates when there are no more events to be executed.

6.1 Performance Measurements

The benchmarks described in Chapter 4 were used to evaluate the performance of the proposed architecture. First we measured the amount of speed improvement over an architecture with global synchronization and without overlapping between matching and selecting-acting within a processor. Then we investigate the effectiveness of the use of associative memories. Finally we obtain estimates for the size of associative memories needed for each one of the benchmarks and for the level of activity in the bus.

6.1.1 Parallel Firing Speedup

To measure the advantages of parallel production firing and of the internal parallelism in each processor, we define a globally synchronized architecture that is very similar to the one proposed in this dissertation, except that it performs global conflict set resolution to implement the OPS5 recency strategy. This synchronized architecture is also very similar to the one suggested by Gupta, Forgy, and Newell [29]. In this architecture, each processor reports the best local instantiation to be fired to the bus controller. The bus controller selects the instantiation whose time tag indicates it to be the latest one to become fireable.

This added decision capability in the bus controller implements the recency strategy to solve the conflict set. The processor selected to fire a production broadcasts all changes in the bus. A processor only selects a new candidate to fire when the matching in the Rete Network is complete. The bus controller waits until all processors report a new candidate to fire. This mechanism reproduces the global synchronization and conflict set generation/resolution present in many of the previously proposed architectures. In order to have a fair comparison, we considered that the synchronized architecture uses an associative memory to store and solve the local conflict sets, and that the bus controller chooses the "winner" in one time step.

Since the synchronized architecture also uses associative memory to store and search the local conflict sets, the comparisons of Figures 6.1 and 6.2 do not reflect the advantages of using such memories in our architecture. We delay this analysis until Section 6.1.2.

Figure 6.1 plots the speed improvement for the benchmarks `life`, `hotel`, `patents`, and `waltz2`. In this and the next section, we will observe a significant difference in performance and memory requirements between this group of benchmarks and the ones based on CTSP (`south`, `south2`, `moun`, and `moun2`). This is due to a gap in complexity between the two groups of benchmarks: the CTSP programs have higher data locality, larger number of productions, and larger data sets. Due to these characteristics, CTSP programs reflect more closely the characteristics encountered in production system applications in industry. The curve names starting with "`s`" indicate measures in the synchronized architecture; the curve names starting with "`a`" indicate measures in the architecture proposed in this paper. All measures compare a given configuration with the speed of a single processor synchronized architecture. For the

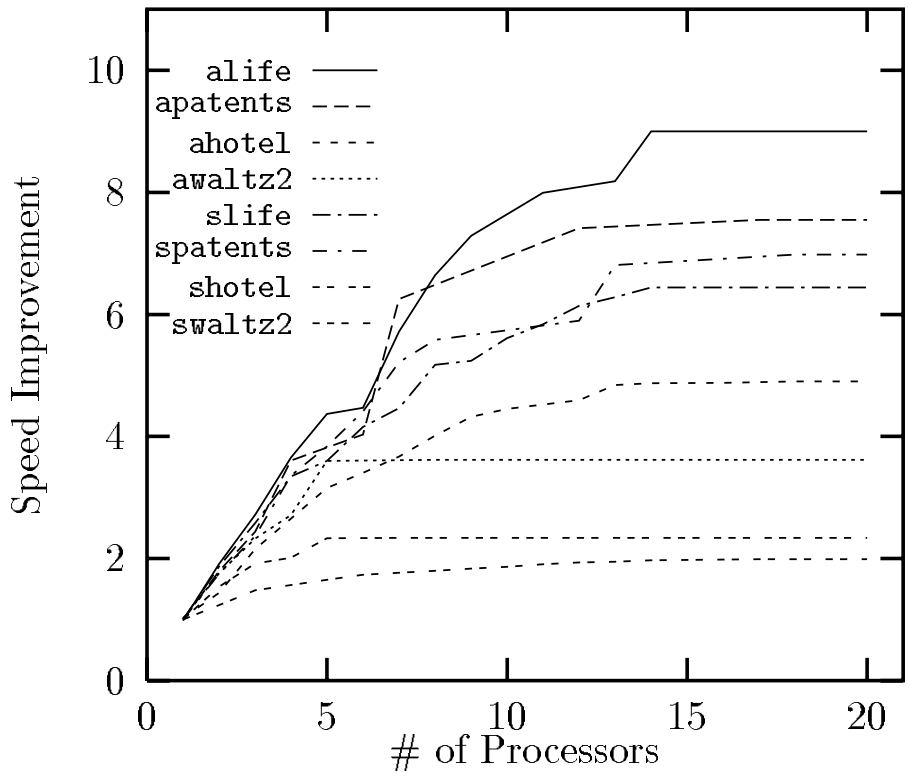

Figure 6.1: Speed improvement obtained by proposed architecture (prefix `a`) and by a reference synchronizing architecture (prefix `s`).

benchmarks presented in Figure 6.1, there is not much distinction between the two architectures when they have a single processor. This indicates that the parallelism between the matching phase and the selecting/execution phase does not result in much speed improvement for these benchmarks. Yet, even with these "toy problems", the parallel firing of productions and the elimination of the global synchronization provides significant speed improvement.

Figure 6.2 shows the comparative performance for the CTSP benchmarks.

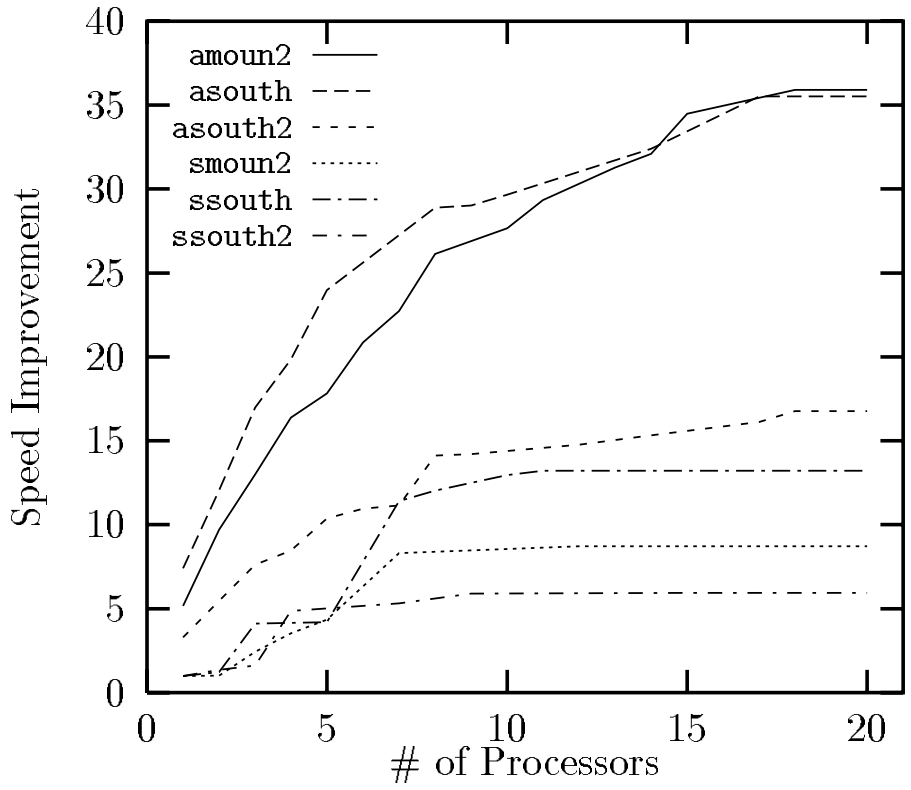

Figure 6.2: Speed improvement obtained by proposed architecture (prefix `a`) and by a reference synchronizing architecture (prefix `s`).

Here, significant speed improvement is observed over the synchronized architecture even for the single processor configuration. This measures the amount of speed that is gained due to the parallelism between matching and selecting/firing. The apparent superlinear improvement in speed in the curves of Figure 6.1 reflects the fact that these curves are showing the combined performance gain due to two different factors: intra and interprocessor parallelism. To obtain the speed improvement due exclusively to parallel production firing, the reader should divide the values in the "`a`" curves by the values in the same curve

for a single processor machine. These results confirm our initial conjecture that the elimination of the global synchronization in a production system allows the construction of machines with significant performance gain.

The number of computing cycles performed by each benchmark in the synchronizing architecture with a single processor machine is presented in Table 6.1. This measurement consider an ideal conflict resolution in which the conflict set is stored in an associative memory in each processor and the global conflict resolution is performed in a single cycle in the bus controller. A benchmark that is not shown in the graphs (`moun`) took 10^{10} cycles to process in the synchronizing architecture. Considering that the actual conflict resolution time is something between 10% and 50% of the processing time [29, 55, 48], we estimate that the same benchmark would take twice as many cycles in a sequential computer. Therefore, this benchmark would be executed in about half an hour in a sequential machine with a 100 MHz processor. A complete study of the scalability of the CTSP benchmarks is presented in appendix B.

Benchmark	Speed Improvement				# of
	mean	σ_s	max	min	cycles
`life`	1.14	0.35	1.28	1.02	5.4×10^8
`hotel`	1.89	1.56	2.50	1.14	2.8×10^6
`patents`	1.02	0.12	1.26	0.87	2.1×10^7
`south`	5.84	9.03	10.18	2.16	1.1×10^8
`south2`	3.53	3.95	5.79	1.77	4.4×10^7
`moun2`	4.90	5.10	8.07	2.87	9.0×10^8
`waltz2`	1.40	0.62	1.56	1.14	8.2×10^7

Table 6.1: Speed improvement over synchronized architecture using the same number of processors.

Another way to compare the two architectures is to measure how much performance gain the proposed architecture has over the synchronized one with the same number of processors. Measurements were made for machines with one through twenty processors. Table 6.1 shows the mean and the variance for the speed improvement obtained with each configuration. It also shows the maximum and minimum speed improvement obtained with any number of processors. Because our architecture implements "eager" production firing without generating a global conflict set, in rare cases, some extra production execution might cause it to be slower than the synchronized architecture (see the minimum speed improvement for `patents`). The gap in performance between the CTSP and the other benchmarks in Table 6.1 indicates that the proposed architecture is very effective in extracting parallelism from PS programs that posses data locality.

6.1.2 Effectiveness of Associative Memories

An associative memory or *content addressable memory* (CAM) is an storage device that retrieves data upon receiving a partial specification of its contents. We adopt Wade's terminology and call a traditional memory accessed by addresses a *reference addressable memory* (RAM) [88]. CAMs are most beneficial for systems in which storage devices are often searched for a cell with a given pattern. The most well known applications of the CAM mechanism are the tag matching in a cache memory and the data checking in a snooping cache or directory. When a CAM receives a request for a piece of data, it searches all positions of the memory and reports the contents of the records that match the specified pattern. Obvious advantages of a CAM over a RAM are the possibility of parallel matching when enough hardware is available to implement it, the

liberation of the processor during memory searches, and reduced traffic between processor and memory [82].

In Chapter 3 we stated that the design of the architecture is based on the premise that the use of CAMs significantly improves the processing speed. In this section we address questions that comes to the mind of an inquisitive computer architect when analyzing the architecture. First, assume a machine configuration in which all memory components are CAM: what would be the impact of replacing one of these CAMs for a RAM? Second, consider a machine in which all memories are RAM: how much speedup would be gained if one of these memory components were to be replaced for a CAM?

To evaluate the speedup obtained by the use of CAMs, we implemented options in the simulator that allow us to specify whether each one of the individual memory components — AFIM, FIM, and PMM — is a CAM or a RAM. If a component is specified as a RAM, the simulator counts the number of accesses performed until the searched data item is found. This number is multiplied by the RAM access time to find the time for that particular access. If a component is specified as a CAM, every access takes the same amount of time.

The effectiveness of a CAM in the architecture depends on the amount of data stored in the memory, the frequency of access, and whether its accesses are in the critical path of execution. Thus, the amount of speedup obtained by a given combination of CAM/RAM memories depends on the production system program that the machine is executing. For a production system program that maintains a large number of productions in the conflict set, the use of CAM for AFIM and FIM might result in a considerable speed improvement. If the conflict set is small, the use of CAM for these memories only improves the speed slightly.

To set up experiments to measure these speedups, we defined two quantities: $Speedup(M, B)$ and $Slowdown(M, B)$. $Speedup(M, B)$ is the amount of speedup that results when the memory component M is replaced for a CAM in a machine that was originally formed only by RAMs. M designates one of the memories component — PMM, AFIM, or FIM — and B is a benchmark program. Equation 6.1 shows how the speedup of PMM is measured.

$$Speedup(\text{PMM}, B) = \frac{Time(\text{PMM}_r, \text{FIM}_r, \text{AFIM}_r, B)}{Time(\text{PMM}_c, \text{FIM}_r, \text{AFIM}_r, B)}, \tag{6.1}$$

where M_r indicates that the memory component M is RAM and M_c indicates that the memory component M is CAM. $Time(\text{PMM}_r, \text{FIM}_r, \text{AFIM}_r, B)$ is the amount of time taken to execute the benchmark B with the architecture configuration specified.

Considering a machine that uses only CAMs, $Slowdown(M, B)$ measures the reduction in speed that would occur if the memory component M were to be replaced for a RAM. Equation 6.2 shows the measurement of the slowdown that results from the transformation of PMM from a CAM to a RAM.

$$Slowdown(\text{PMM}, B) = \frac{Time(\text{PMM}_r, \text{FIM}_c, \text{AFIM}_c, B)}{Time(\text{PMM}_c, \text{FIM}_c, \text{AFIM}_c, B)}. \tag{6.2}$$

For a given benchmark program the amount of speedup obtained by using CAM memories varies with the number of processors used in the architecture. Table 6.2 presents the average speedup for machines with one up to twenty processors. In practical designs, CAMs might be slower than RAMs: either because they are constructed with older technology, or because they need to use more silicon area for the comparator circuits. To account for these factors we

introduce a *technology factor* T that indicate how much slower a basic operation such as the reading or writing of a single data element was considered in this comparison. Table 6.2 shows measures for a machine with CAMs with the same speed as the RAMs ($T = 1$) and for a machine with CAMs that are four times slower ($T = 4$) than the RAMs. Observe that there is no significant difference in speedup between the two measures, indicating the advantage of the use of CAMs, even if they are slower than RAMs.

Benchmark	T	PMM		FIM		AFIM		All
Bench	T	*Speed*	*Slow*	*Speed*	*Slow*	*Speed*	*Slow*	*Speed*
`hotel`	1	3.0	29.3	1.0	1.6	1.6	13.5	45.5
`hotel`	4	3.0	30.1	1.0	1.6	1.5	13.6	45.3
`life`	1	2.8	2.1	1.3	1.0	1.6	1.2	3.4
`life`	4	2.8	2.1	1.3	1.0	1.6	1.2	3.4
`moun2`	1	3.3	4.9	1.0	1.0	1.8	2.5	8.5
`moun2`	4	3.3	4.9	1.0	1.0	1.7	2.5	8.5
`patents`	1	1.9	1.6	1.0	1.0	1.4	1.2	2.3
`patents`	4	1.9	1.6	1.0	1.0	1.4	1.2	2.3
`south2`	1	3.4	10.0	1.0	1.1	1.4	4.3	14.8
`south2`	4	3.3	10.2	1.0	1.1	1.5	4.4	14.9
`waltz2`	1	1.8	1.4	1.0	1.0	1.9	1.6	3.0
`waltz2`	4	1.8	1.5	1.0	1.0	1.9	1.6	3.0

Table 6.2: Speedup due to use of CAMs[2].

Table 6.2 shows the speedup and the slowdown due to each piece of associative memory for each one of the benchmarks presented in Chapter 4. The last column shows the speedup that compares a configuration with all three memories associative against one in which all three memories are RAM. Table 6.2 shows that replacement of just one memory for a CAM results in quite low

speedup. This limited speedup is result of the slow operation of the RAMs in the machine. Only when all three memories are made CAMs, the processing speed shows considerable improvement. The numbers in the slowdown columns show that the use of RAM in PMM or AFIM alone might cause significant reduction in speed. Both experiments show that the use of CAM for FIM is not very important. Overall, these results confirm our initial conjecture that the use of CAMs can provide considerable speedup in production system architectures.

6.1.3 Associative Memory Size

The next question that the inquisitive computer architect must ask is: how large do these associative memories need to be? The simulator has an option to report the "crest"[3] of each memory component in any given run. Table 6.3 shows the maximum and the average crest over machines with up to twenty processors. The average crest is the average of the largest memory needed for each machine configuration. The maximum crest indicates the minimum memory size needed to run that specific benchmark. Observe that for some memory/benchmark the average crest is several times smaller than the maximum crest (see AFIM in `moun2` and PMM in `waltz2`). If memory size becames a concern in the construction of the machine, a RAM can be used to contain overflow. The absence of a direct correlation between the size of the memory crest and the speedup and slowdown shown in Table 6.2 reflects the fact that the processing speed is not solely dependent on the amount of data stored in each memory: it also depends on the frequency and time of access of these memories.

[2]Each number is an average of 20 values, obtained for systems with 1 through 20 processors.

[3]The *crest* of a memory component is the maximum amount of data stored in that memory component in any processor of the machine for a given benchmark and a specified number of processors.

Bench-	PMM		FIM		AFIM		FIM(sync.)	
mark	Max	Ave	Max	Ave	Max	Ave	Max	Ave
`hotel`	3200	1436	395	216	1030	699	3580	1178
`life`	2877	2643	690	584	3313	1472	23030	8787
`moun2`	27899	23303	2580	727	15634	3042	313400	46747
`patents`	776	739	605	179	1549	449	1410	426
`south2`	4788	2822	350	95	1159	611	47205	8414
`waltz2`	3573	1109	1250	870	2797	1688	5785	3299

Table 6.3: Maximum and average "crest" for memory size (bytes).

The speed comparison with the synchronized architecture presented in section 6.1.1 considered that both architectures used associative memory to store and search the conflict set. The average and the maximum crests of the associative memories for the synchronized architecture are presented in the rightmost columns of Table 6.3. Observe that for most of the significant benchmarks, the synchronized architecture needs much larger memory. For the CSTPs benchmarks (`moun2` and `south2`) the maximum crest in the synchronized architecture was ten times larger than in the architecture proposed in this paper. This evidences that the "eager firing" mechanism also reduces the demand for memory.

6.1.4 Use of Bus

A legitimate concern about any bus-based parallel architecture is the limitation of a bus as a broadcasting network. In Chapter 3 we conjectured that bus bandwidth is not a limitation in the architecture proposed. Table 6.4 presents the measurements for the percentage of time that the bus is busy assuming that bus bandwidth is the same as that of local memory. These measures include the

Benchmark	Bus Utilization(%)		
	4 proc.	8 proc.	16 proc.
hotel	10.9	20.9	23.7
life	0.83	1.38	2.02
moun2	2.25	3.83	4.71
patents	0.68	0.89	1.08
south2	4.97	8.31	9.72
waltz2	1.36	1.79	1.76

Table 6.4: Percentage of time that the bus is busy.

arbitration time and the token broadcasting time. Observe that technological limitations would have to render the bus much slower than the memories before the bus speed becames a concern in this architecture.

Chapter 7

Rete Network with Multiple Functional Units

The architectural model presented in chapter 3 and evaluated in chapter 6 assumed that the matching of the knowledge base with production antecedents was performed by a single functional unit Rete Network. A closer look at Production System execution on this model reveals that the bottleneck is in the processing of tokens in the β-nodes of the Rete Network rather then at the associative memories or in the communication bus. This motivates the use of a multiple functional unit Rete Network within each processor. Because the fraction of time spent in α-nodes is very small, this study concentrates on the effect of parallelizing the execution of tokens at the β-node level while using a single functional unit specialized for α-node matching. We refer to a functional unit that executes tokens in β-nodes as a β-unit. An α-unit is a functional unit that executes tokens in α-nodes.

The proof of correctness established in chapter 3 assumed that tokens were

processed in the Rete Network exactly in the same order in which they arrived. The use of multiple β-units adds complexity to the problem of synchronizing the matching of tokens in the Rete Network with the firing of productions in the Instantiation Firing Engine. To ensure a correct operation of the multiple β-unit architecture, two synchronizing structures are added: an *In-Order Buffer* and a *Re-Order Buffer*. This chapter presents the organization for this new Rete Network scheme, the solution for the synchronization problem, and some simulation results that indicate that a considerable improvement in the machine performance is obtained with the use of a modest number of β-units.

7.1 Multiple β-Unit Rete Network

In the organization presented in Figure 7.1, the α-unit is responsible for processing tokens in α-nodes. Each incoming token is sequentially tested against all α-nodes of the network. Whenever there is a match with an α-node, the α-unit replicates the token and attach to each copy the identity of one of the β-nodes that are successors of the α-node where the match occurred. These new tokens are then forwarded to the β-units. Therefore the processing of a single token in the α-unit may result in the generation of many tokens to be inserted in the *external* β-queue. Note that β-units also share an *internal* input queue for tokens generated by β-nodes.

To reduce the average time that tokens spend in the Rete Network, a token placed in the external queue is processed only when the internal queue is empty. A free β-unit will take the first token from the queue, read the memory locations corresponding to the β-node to which the token is destined, execute all the β-tests and then place the newly generated tokens at the end of the internal queue.

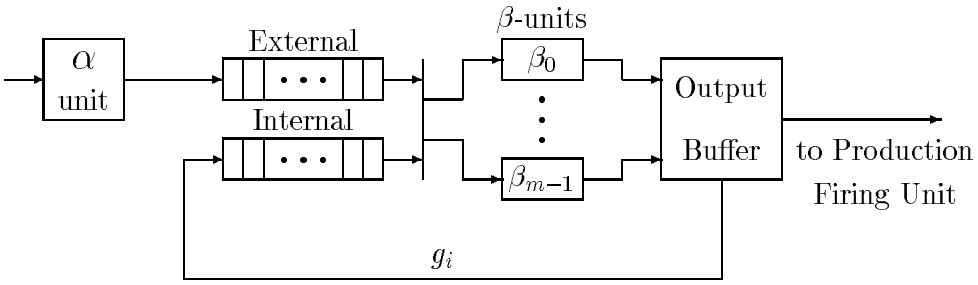

Figure 7.1: Rete Network with m β-Units

When a token destined for a P-node is produced by a β-unit it is placed in the Output Buffer to be forwarded to the production firing unit.

This study does not consider the use of multiple α-units. Therefore, the amount of speedup in the Rete Network, $S_{Rete}(m)$, according to Amdahl's Law [37], is limited by the amount of time spent in the non-parallelized portion of the algorithm.

$$S_{Rete}(m) = \frac{T_\alpha + T_\beta(1)}{T_\alpha + T_\beta(m)}, \tag{7.1}$$

where m is the number of β-units in the architecture, T_α is the average amount of time spent in α nodes, and $T_\beta(m)$ is the average amount of time spent in β nodes when m identical β-units are used. The maximum speedup that can be obtained with such an architecture is given by

$$\lim_{m\to\infty} S_{Rete}(m) = 1 + \frac{T_\beta(1)}{T_\alpha}. \tag{7.2}$$

7.2 Synchronization Issues

Because multiple tokens are simultaneously processed in the β-units, a careful consideration of the order in which the results appear at the output is nec-

essary to guarantee correct results. The organization presented in Figure 7.1 allows out-of-order execution of tokens in the β-units and might cause token deletions to be processed before the corresponding additions. Mechanisms are necessary to ensure that out of order processing of tokens do not lead to wrong results. Simple solutions such as allowing only tokens originated from a single external token to proceed to the β-units are too conservative and might eliminate the advantages of a multiple beta unit organization. It is necessary to create a synchronization mechanism that imposes minimum overhead and guarantee correct results in every situation.

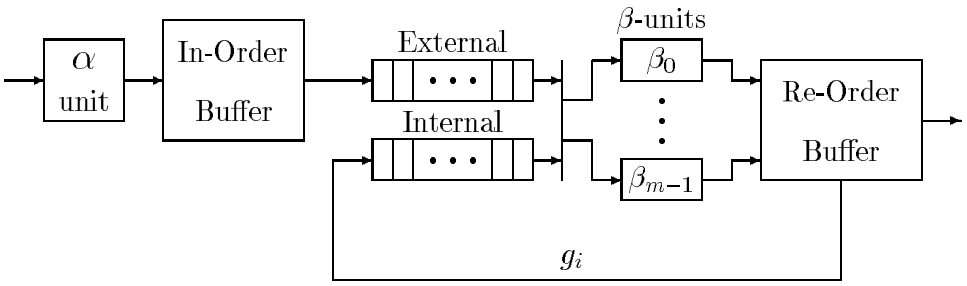

Figure 7.2: Synchronizing Buffers in Rete Network with m β-Units

The solution shown in Figure 7.2 uses an *In-Order Buffer* (IOB) and a *Re-Order Buffer* (ROB). IOB prevents conflicting tokens from entering the β-unit queues simultaneously. ROB assures that changes to the conflict set are delivered to the instantiation firing unit in the same order as the arrival of the external tokens that caused these changes. The Output Buffer of Figure 7.1 is included in the representation of the Re-Order Buffer in Figure 7.2. The idea of using buffers to overcome synchronization problems is borrowed from superscalar and superpipelined architectures [15, 75, 77].

7.3 In-Order Buffer

The purpose of the In-Order Buffer is to prevent two conflicting tokens from proceeding to the β-units while allowing non-conflicting tokens to be processed without delay. Two tokens are *conflicting* if one of them enables and the other disables the same production. To decide whether two tokens are conflicting, IOB uses the type of action specified by the tokens and the identification of the β-node to which they are destined.

Before the organization of IOB is presented, it is necessary to discuss the internal organization of the Rete Network. By convention, the children of a Rete Network node are labeled as "left" and "right" children. All α-nodes are left children of the root node. The root node is the only parent of each α-node. Each β-node is associated with an antecedent of a production. A sign associated with the β-node indicates whether this antecedent is negated. We will notate the sign of the i^{th} β-node with $\mathcal{S}(\beta_i)$. A β-node has two inputs (or parents). The left parent of a β-node is always the α-node that performs the α testing in the antecedent associated with the β-node. If the antecedent associated with a node β_i is negated, $\mathcal{S}(\beta_i) = -1$, otherwise $\mathcal{S}(\beta_i) = +1$.

The right parent of a β-node might be another β-node or an α-node. If it is an α-node, it must be the first antecedent of a production and therefore cannot be negated. If the right parent is a β-node, it is the result of a join operation of at least two independent antecedents of a production and cannot be negated either.

Each production R_a has a corresponding P-node in the Rete Network. A production that has a single antecedent has no need for join operations and no β-nodes. In this case the left son of an α-node is a P-node.

We say that a β-node β_i *belongs* to a production R_a, notated by $\beta_i \in R_a$ if there is a line of successors (or a path of β-nodes) in the Rete Network that connects β_i to the P-node of R_a. In other words, if any token produced by β_i can eventually lead to the addition or removal of an instantiation of R_a, then $\beta_i \in R_a$. Because parts of the Rete Network might be shared among distinct productions, a β-node might belong to more than one production.

At compile time, an analysis of the Rete Network is performed by transversing the network from the α-nodes to the P-nodes following the left son links. This results in the construction of a two dimensional array with information about the relationship among the beta nodes. This array has two bits in each position and stores the function $\mathcal{C} : \mathcal{B} \times \mathcal{B} \rightarrow \{0, +1, -1\}$, where $\mathcal{B}$ is the set of all β-nodes in the Rete Network.

$$\mathcal{C}(\beta_i, \beta_j) = \begin{cases} 0 & \text{if } \nexists R_a \in P_k / \beta_i \in R_a \text{ and } \beta_j \in R_a \\ +1 & \text{if } \exists R_a \in P_k / \beta_i \in R_a \text{ and } \beta_j \in R_a \text{ and } \mathcal{S}(\beta_i) = \mathcal{S}(\beta_j) \\ -1 & \text{if } \exists R_a \in P_k / \beta_i \in R_a \text{ and } \beta_j \in R_a \text{ and } \mathcal{S}(\beta_i) \neq \mathcal{S}(\beta_j) \end{cases}$$

The array formed with the values of $\mathcal{C}(\beta_i, \beta_j)$ is used to identify pairs of tokens that enable and disable the same production. The function of IOB is to prevent such tokens from being executed concurrently in the β-units.

Tokens arriving at the Rete Network are classified according to the action that they produce in the specified working memory element: add, delete, or modify. Because a modify token is pre-processed in the α-unit producing an add and a delete token, these are the only types of actions to be processed in β-nodes. We define the function $\mathcal{A} : \mathcal{T} \rightarrow \{+1, -1\}$, where $\mathcal{T}$ is the set of all possible tokens to be processed in any β-node.

$$\mathcal{A}(T_i) = \begin{cases} -1 & \text{if } T_i \text{ is a delete token} \\ +1 & \text{if } T_i \text{ is an add token} \end{cases}$$

After the processing in the α-nodes, a token is identified by a pair (T_i, β_i) that contains the token identification T_i and the identification of the β-node that will process the token. The decision of whether two tokens are conflicting takes into account both values. Suppose that a token (T_i, β_i) is already in IOB when a second token (T_j, β_j) arrives. The IOB decision to hold the second token until the processing of the first one is completed or to forward the new token immediately is based on the function $Conflict(i, j)$.

$$Conflict(i, j) = \mathcal{A}(T_i)\, \mathcal{A}(T_j)\, \mathcal{C}(\beta_i, \beta_j) \tag{7.3}$$

If $Conflict(i, j)$ is positive or equal to zero, there is no conflict between the two tokens and the second token to arrive in IOB can proceed to the external queue even if the processing of the first one has not been completed. Observe that if the tokens are destined to β-nodes that are not part of the same production, the value of $Conflict(i, j)$ is zero, independent of the types of action in the tokens. Two tokens destined to the same β-node are conflicting if their actions are different. Therefore, by definition $\mathcal{C}(\beta_i, \beta_i) = +1$ for any node β_i.

The In-Order Buffer is an associative memory with the organization shown in Figure 7.3. It works as a queue with some additional decision logic. When a token arrives in the buffer, a single associative search identifies whether there is another token already in the buffer with which the new token conflicts. If no conflicting tokens are found in the buffer, the token is placed at the end of the buffer with NULL value in its *conflicts_with* field. Simultaneously, the token is

Presence	T_i	β_i	$A(T_i)$	has_conflict	conflicts_with
⋮	⋮	⋮	⋮	⋮	⋮

Figure 7.3: Organization of In-Order Buffer (IOB)

placed in the external queue to be processed by the β-units. If more than one conflicting token is found, lets assume that the one closer to the end of the buffer is stored in position k of the buffer. The newly arrived token is stored in the end of the buffer with the value k stored in its *conflicts_with* field. The single bit field *has_conflict* of the position k of the buffer is set to 1 indicating that the completion of the processing of the token in this position is been awaited by another token. Because there are no tokens waiting for the completion of the newly arrived token, its *has_conflict* field is initially reset.

IOB_Arrival(T_i, β_i,tail)

1 IOB(tail).$T_i \leftarrow T_i$
2 IOB(tail).$\beta_i \leftarrow \beta_i$
3 IOB(tail).*presence* $\leftarrow 1$
4 $k \leftarrow \max_j\{j \ / \ (T_j, \beta_j) \in$ IOB and $Conflict(k, j) < 0\}$
5 **if** $k = -1$
6 **then** IOB(tail).*conflicts_with* $\leftarrow$ NULL
7 **forward** (T_i, β_i) to β-units
8 **else** IOB(tail).*conflict_with* $\leftarrow k$
9 IOB(k).*has_conflict* $\leftarrow 1$

In the algorithm that represents the sequence of steps for the arrival of a token in IOB, the argument tail is the first empty position at the end of the buffer. Step 4 is the associative search for possible conflicting tokens already in the buffer.

The processing of a token that is recorded in IOB is completed when the token and all its successors have been processed. This completion is signaled by the Re-Order Buffer. When a token has been completely processed, if its *has_conflict* field is reset, the token is removed from IOB just by reseting the *presence* field. If the *has_conflict* field is set, an associative search in the *conflicts_with* field of the buffer identifies all tokens that were waiting for this completion. Suppose that one of these tokens is found at position p of the buffer. The token at p might still conflict with some other token that arrived before the token now completed, but that has not been processed yet. Therefore, it is necessary to perform another associative search in the buffer to identify whether any other token ahead of p conflicts with p. If one such token is found at position q, the value of the *conflicts_with* field of p is changed to q, the field *has_conflict* of p is set, and the token in p remains waiting. If the token in p does not conflict with any other token ahead of itself, its *conflicts_with* field is set to NULL and the token is placed in the external queue.

Observe that the value stored in the field *conflicts_with* of a token that has many conflicting tokens in the IOB is always the one closest to the end of the buffer. It is likely that all other conflicting tokens will have left when this token leaves. Therefore the second search for further conflicting tokens will find none in most of the cases. Also, the presence of the single bit field *has_conflict* in the IOB prevents the execution of an associative search for tokens that do not have any conflicting token waiting for them.

IOB_Departure(T_i, β_i)

1 $k \leftarrow$ position of (T_i, β_i) in IOB
2 **if** IOB(k).*has_conflict* $= 1$
3 then **foreach** $p \in \{r$ / IOB(r).*conflicts_with* $= k\}$
4 **do** $q \leftarrow \max_j\{j$ / $(T_j, \beta_j) \in$ IOB and
$Conflict(p, j) < 0$ and $j \prec q\}$
5 **if** $q = -1$
6 **then** IOB(p).*conflicts_with* $\leftarrow$ NULL
7 **forward** (T_p, β_p) to β-units
8 **else** IOB(q).*has_conflict* $\leftarrow 1$
9 IOB(p).*conflicts_with* $\leftarrow q$

In the algorithmic presentation of the procedure for departure from IOB, steps 1, 3, and 4 involve an associative search. The relationship $j \prec q$ indicates that the token at position j precedes the token at position q in the buffer.

7.4 Re-Order Buffer

The Re-Order Buffer has the function of identifying the departure of a token from the system and guaranteeing that changes to the conflict set are delivered to the instantiation execution unit in the same order in which tokens arrived at the α-units. The ROB is actually comprised of two buffers, one for "α-tokens" and one for "β-tokens". In this context, an α-token is a token that arrived at the α-unit. The processing of an α-token might produce a number of β-tokens to be placed in the external queue. Once a token arrives at the Rete Network, it is assigned a unique identification. Every token generated internally also receives

a unique identification. The α-tokens are stored in the α-buffer according to its arrival order. Whenever a β token is generated from an α-token, it is placed in the β-buffer and the counter in the α-buffer is increased.

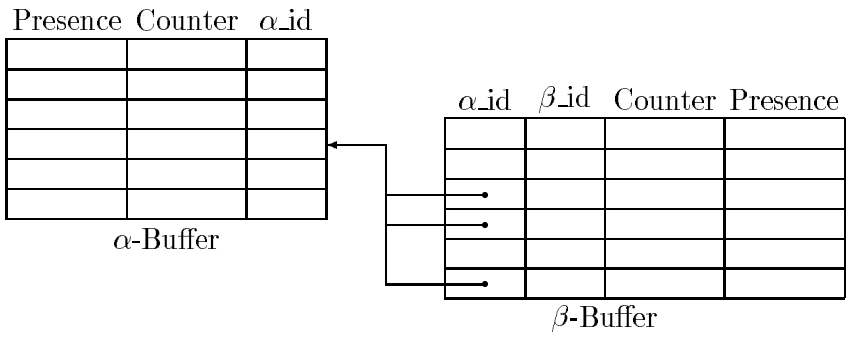

Figure 7.4: Organization of Re-Order Buffer (ROB)

The execution of a β-token might produce further tokens. We call these tokens generated at the β-units *internal tokens*. For all practical purposes there is no distinction between a β-token and an internal token other than that the former is produced in an α-unit while the later is produced in a β-unit. The identification of a β-token is carried along by all its successors. Whenever a new internal token is produced, the counter corresponding to its original β-token is incremented in the β-buffer. When an internal token is processed, this counter is decremented. Whenever a β-buffer counter reaches zero, the β-token is removed from the buffer, the IOB is notified of the token departing, and the respective counter in the α-buffer is decremented.

When an α-buffer counter reaches zero, the last β-token generated by that α-token is leaving the system. The token is removed from the α-buffer. Any subsequent token that has a counter equal to zero can also leave. The memory that stores changes to the conflict set is searched for changes that are associated

with these tokens. After these changes are delivered to the instantiation firing unit, the token is removed from the α-buffer.

7.5 Experimental Results

Tables 7.1 and 7.2 present the speedups obtained by increasing the number of processors and the number of beta units in the architecture of Figure 3.2. All measures presented represent architectures with a separate unit for α-node processing. On the top of each column is the number of β-units in the architecture.

The results clearly show the advantages of using multiple β-units. In general, given limited resources, it is preferable to use fewer, more powerful processors than a larger number of less powerful ones. For example, a 2 processor system with five β-units per processor performs better than a five processor system with two β-units per processor.

There are few situations in which increasing the number of beta units or the number of processors (or both) results in a reduction of speedup. This happens because: (1) the architecture utilizes a partially informed selection mechanism that might result in the firing of extra productions; (2) for a given configuration in an specific benchmark, the firing engine might select productions and wait for the possession of the bus just to find out that the selected production is no longer fireable and needs to be aborted. Therefore, Tables 7.1 and 7.2 should be examined for the trends with the increasing of the number of processors and beta units, and not for any specific value.

The higher speedup delivered by a faster, multiple functional unit Rete Network confirms some early observations in production system research that the

Bench.	# Proc	# of Beta Units						
		1	2	3	5	7	8	10
patents	1	1.00	1.98	2.88	4.64	6.32	6.93	8.44
	2	1.54	3.02	4.50	7.30	9.78	10.62	13.09
	5	2.70	5.28	8.03	13.45	18.71	20.83	23.48
	7	4.02	7.90	11.41	19.71	22.13	22.83	23.68
	10	5.34	10.56	14.79	21.28	23.63	24.07	24.55
	15	6.20	12.82	18.55	22.07	23.40	23.79	24.49
	20	7.05	14.23	19.45	22.22	23.49	23.98	24.61
waltz2	1	1.00	1.99	2.87	4.03	4.80	5.05	5.42
	2	1.73	3.17	4.05	5.07	5.61	5.78	6.03
	5	3.46	4.64	5.33	5.99	6.33	6.44	6.55
	7	3.46	4.64	5.33	6.00	6.35	6.43	6.57
	10	3.46	4.64	5.31	5.99	6.35	6.43	6.55
hotel	1	1.00	1.32	1.36	1.36	1.36	1.36	1.36
	2	1.73	1.78	1.79	1.80	1.80	1.80	1.81
	5	3.70	3.77	3.75	3.77	3.79	3.79	3.79
	7	5.17	5.59	6.07	6.15	5.69	6.08	6.25
	10	5.87	6.96	6.64	6.96	6.76	6.90	6.83
	15	5.95	7.17	7.16	7.22	7.25	7.30	7.28
	20	5.95	9.15	8.95	9.00	8.82	9.08	8.87

Table 7.1: Speedups of a concurrent production system with multiple β-functional units.

matching phase of the execution constitutes a bottleneck. However, for some benchmark, the improvement obtained by a faster Rete Network saturates with a rather small number of functional units, evidencing the need for multiple processors.

Bench.	# Proc	# of Beta Units						
		1	2	3	5	7	8	10
south	1	1.00	1.97	2.89	4.63	6.00	6.80	7.99
	2	1.70	3.26	4.76	7.25	9.20	10.24	11.65
	5	3.07	6.06	9.94	11.88	15.91	16.05	18.38
	7	4.07	6.25	9.08	12.86	17.80	19.15	20.22
	10	3.41	6.71	10.05	15.63	18.42	19.58	20.73
	15	4.22	8.03	11.29	16.07	18.77	20.54	21.21
	20	4.38	8.50	10.32	15.12	20.80	21.60	22.27
south2	1	1.00	1.96	2.86	4.34	5.62	6.20	7.12
	2	1.75	3.43	4.84	6.69	8.08	8.66	9.94
	5	3.06	5.59	7.73	10.85	12.70	13.03	13.75
	7	3.61	6.86	9.45	12.98	15.50	16.26	16.95
	10	4.21	7.60	10.22	12.87	14.75	15.06	15.03
	15	4.45	8.33	11.07	14.94	17.12	16.49	18.05
	20	4.58	8.44	11.48	15.27	17.14	17.43	17.93
moun2	1	1.00	1.94	2.81	4.40	5.82	6.44	7.58
	2	1.85	3.41	4.97	7.31	9.11	9.87	10.90
	5	3.43	6.53	8.93	12.67	15.03	15.88	16.29
	7	3.98	7.30	9.74	13.29	14.77	14.99	15.18
	10	5.02	9.13	12.30	16.42	19.13	19.89	20.23
	20	7.24	12.98	18.18	25.84	30.43	32.09	33.91

Table 7.2: Speedups of a concurrent Production System with multiple β-functional units.

Chapter 8

Analytical Model

Have you ever encountered a queue,
In which Poisson arrivals accrue?
In Statistics, I'm told
This assumption can hold...
... But it sure sounds more fishy than true!

(Ben W. Lutek) as quoted in [5].

Performance evaluation can be accomplished through measurement, simulation, and analytic modeling [41]. Measurement consists of observing actual values for specified parameters in an existing system. Simulation consists in creating a model for the behavior of a system, writing a computer program that reproduces this behavior, feeding the program with an appropriate sample of the workload of the actual system, and computing selected parameters of interest. In analytic modeling a mathematical model of the system is created and its solution provides the performance evaluation. The problem with ana-

lytic modeling is that few detailed mathematical models can actually be solved. The good news is that even simplified models often deliver remarkably good approximations for performance estimates [41].

Analytical modeling is a powerful and underused tool in the study of production system machine performance. Yukawa *et al.* [93, 44] construct an analytic model for the performance of Rete Network based on basic notions of probability, utilizing a "back of the envelope" method similar to the one largely employed by Cragon [15] for the analysis of superpipelined and superscalar processors. Wang et al. [89] constructed an analytical model to measure performance of parallel-rule firing production systems based on a transaction model.

In this research we use what Kant [41] describes as an "hybrid modeling" method. We use the simulator described in Chapter 6 to obtain the parameters necessary to evaluate the performance of the Rete Network with multiple β-units described in Chapter 7. Based on these parameters, we construct an analytic model for the processing of tokens in the Rete network. This model allows the estimation of the improvement in the speed of the Rete Network when m β-units are used.

This chapter presents an analytical model based on queueing theory to estimate the speedup obtained by the use of multiple β-units in the implementation of the Rete Network. For clarity, we start with a single β-unit system, introducing the generating function technique that is used to solve the analytic model [90, 45, 41]. In section 8.3.2 we develop a model for a system with m servers, and verify that it reduces to the single server model when $m = 1$.

8.1 Single β-Unit Architecture

In this section we consider an organization with a single β-unit and a single α-unit. In section 8.2 the model is extended to represent an organization with multiple β-units. A number of simplifications were necessary to construct the analytical model for the Rete Network with multiple functional units. Delays due to the synchronizing *In-Order Buffer* and *Re-Order Buffer* are not considered in the analytical model. Also we consider an organization with a single queue for both external and internal tokens. Figure 8.1 shows the single β-unit organization considered for the analytical model.

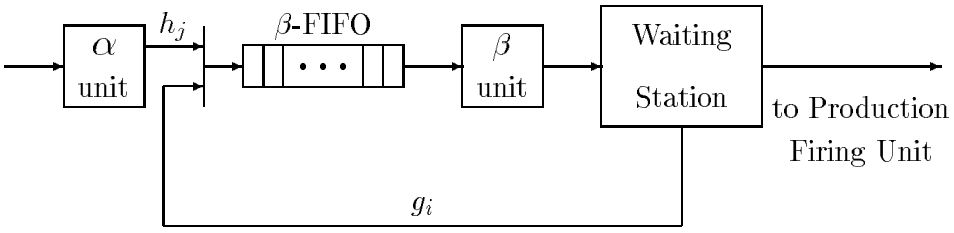

Figure 8.1: Single β-Unit System

Whenever a token matches an α-node while being processed in the α-unit, a number of tokens are produced to be delivered to the β-FIFO. This group of tokens is called a *bulk*. The probability that a bulk has j tokens is measured by h_j. The model does not consider empty bulks, i.e., an arrival occurs when at least one token arrives in the β-FIFO. Therefore $h_0 = 0$.

Whenever free, the β-unit will take the first token from the queue, read the memory locations corresponding to the β-node to which the token is destined, execute all the β-tests and then place the newly generated tokens at the end of the β-FIFO. If the processing of a token in a given β-node produces a new tokens, and that β-node has b β-node successors, we consider that ab new tokens

were generated and placed at the end of the queue. The probability that i new tokens are generated when a token is processed in a β-node is given by g_i. Also, any new instantiation at a terminal node is sent to the production firing unit that will select one (or more) production to be fired.

For the construction of the analytical model, we consider that there is a "waiting station" at the β-unit output that allows it to hold all the tokens generated until the end of the execution of the current token, and then adds all of them at once to the queue. Although this assumption may not reflect the actual behavior of a physical machine[1], it simplifies the analytical model.

8.2 Multiple β-Unit Architecture

The organization considered for the construction of an analytical model for the Rete Network with multiple β-units is shown in Figure 8.2. The only difference with the single β-unit architecture is the number of β-units used to process tokens in β-nodes.

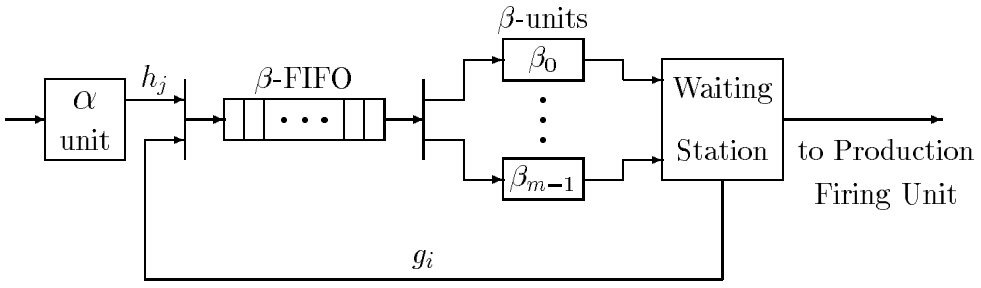

Figure 8.2: System with m β-Units

[1] In an actual machine the β-unit would place tokens in the queue as they are produced.

8.3 Analytical Model

In this model, the arrival of a bulk of tokens in the β-queue from an α-node is called an *external arrival*. The arrival of a group of tokens from the *waiting station* into the β-FIFO is an *internal arrival*. The moment at which a β-unit finishes processing a token and places all the newly generated tokens (if any) in the β-FIFO is called *departure*. To develop the queueing theory model, we make the following assumptions:

- The β-FIFO has infinite capacity.
- The overhead of placing tokens in the β-FIFO is zero.
- The external arrival of tokens is a Poisson process.
- The processing time for tokens in the β-nodes follows a Poisson process.

The main problem with the above assumptions is that events that occur in the machine are not independent of each other. In fact they not only depend on the specific production system being executed, but also change as the execution of it proceeds. However, the simplified model obtained still produces a good estimate for the amount of time that a token spends in the system, and thus for the reduction in the average time spent by a token in an m β-unit system compared with a single β-unit system.

In the next section we present the analytical model for the organization with a single β-unit. In section 8.3.2 we introduce the model for the multi β-unit architecture. Appendix C presents a detailed derivation for both models.

8.3.1 Single β-Unit Model

The infinite Markov chain that represents the single β-unit system is presented as a state-transition-rate diagram in Figure 8.3. The circles represent states and the arcs represent state transitions. The value associated with each arc indicates the transition rate. In this representation, the system is in state k if there are k tokens to be processed in the system, including the token currently being processed. Let g_i be the probability that the processing of a token in the β-unit produces i new tokens, h_j be the probability that an external arrival brings into the system a bulk with j tokens, λ_β be the external arrival rate into β-nodes, and μ_β be the processing rate in β-nodes.

For clarity, not all transitions are shown in Figure 8.3. Consider the transitions from and to state k. Any state $i < k$ can reach state k in a single transition. Except for state 0, every state has a self-transition with transition rate $\mu_\beta\, g_1$ indicating that upon finishing processing a token, the β-unit generated one token, therefore the number of tokens in the system remains the same. The transition from state k to state $k-1$ indicates that the processing of the current token produced no new tokens.

A transition out of state k might occur due to an external arrival or due to the completion of the processing of a token in the β-unit. Whenever $i>1$ tokens are generated by the β-unit, the system changes from state k to state $k+i-1$ upon the completion of β-unit processing. If a bulk of size j arrives from the α-unit while the system is at state k, the system moves to state $k+j$.

The only possible transitions out of state 0 are due to the external arrival of a bulk of tokens. The processing rate at state 0 is equal to zero. This reflects the fact that no token can be processed when there are no tokens in the system.

Figure 8.3: State Diagram for a Single β-Unit System

We are interested in the steady state behavior of the system. At steady state, the input flow must equal the output flow in each state. Therefore, we can write difference equations for the system.

$$(\lambda_\beta + \mu_\beta \sum_{i=0}^{\infty} g_i)\, p_k = \mu_\beta\, g_0\, p_{k+1} + \lambda_\beta \sum_{j=0}^{k-1} p_j\, h_{k-j} + \mu_\beta \sum_{i=1}^{k} p_i\, g_{k-i+1}, \quad (8.1)$$

where p_k is the probability of k tokens being in the system, including the token being processed.

By definition, the sum of all probabilities g_i is equal to unity. Using $\sum_{i=0}^{\infty} g_i =; 1$, equation 8.1 can be written as follows:

$$(\lambda_\beta + \mu_\beta)\, p_k = \mu_\beta\, g_0\, p_{k+1} + \lambda_\beta \sum_{j=0}^{k-1} p_j\, h_{k-j} + \mu_\beta \sum_{i=1}^{k} p_i\, g_{k-i+1}. \quad (8.2)$$

Examining the transitions into and out of state 0, we can write the boundary equation 8.3.

$$\lambda_\beta \sum_{j=0}^{\infty} h_j\, p_0 = \mu_\beta\, g_0\, p_1 \quad (8.3)$$

But the sum of all probabilities h_j is also equal to unity. Therefore the boundary equation can be simplified to equation 8.4

$$\lambda_\beta \, p_0 \; = \; \mu_\beta \, g_0 \, p_1 \tag{8.4}$$

From equations 8.2 and 8.4, using Z-transforms (see Appendix C), we obtain the generating function for the system with a single β-unit:

$$P(z) \; = \; \frac{\mu_\beta \, (1 \; - \; \rho_1)(z \; - \; G(z))}{\lambda_\beta \, z \, [1 \; - \; H(z)] \; + \; \mu_\beta \, [z \; - \; G(z)]}, \tag{8.5}$$

where $P(z)$, $G(z)$, $H(z)$, and ρ_1 are defined as

$$P(z) \; = \; \sum_{k=0}^{\infty} \, p_k \, z^k, \tag{8.6}$$

$$G(z) \; = \; \sum_{k=0}^{\infty} \, g_k \, z^k. \tag{8.7}$$

$$H(z) \; = \; \sum_{k=0}^{\infty} \, h_k \, z^k. \tag{8.8}$$

$$\rho_1 \; = \; \frac{\rho_0 \, \overline{H}}{(1 \; - \; \overline{G})} \; = \; \frac{\lambda_\beta \, \overline{H}}{\mu_\beta \, (1 \; - \; \overline{G})}, \tag{8.9}$$

$$\overline{G} \; = \; \lim_{z \to 1} G^{(1)}(z) \; = \; \sum_{i=0}^{\infty} \, i \, g_i \tag{8.10}$$

$$\overline{H} \; = \; \lim_{z \to 1} H^{(1)}(z) \; = \; \sum_{i=0}^{\infty} \, i \, h_i \tag{8.11}$$

The *utilization factor* ρ_0 is a ratio between the external arrival rate λ_β and the β-unit processing rate μ_β; ρ_0 represents the utilization rate in an M/M/1 system, i.e., a system with no bulk external arrivals and where new tokens are not generated at the β-units. The value ρ_1 is the utilization rate of a single β-unit system with bulk arrivals and "feedback", i.e., a system in which the β-units generate new tokens. This rate must be positive, therefore the *first moment* of g_i, $\overline{G}$ must be less than one. Also, for stability, we must have $\rho_1 < 1$

what implies that

$$\frac{\lambda_\beta}{\mu_\beta} < \frac{(1-\overline{G})}{\overline{H}} \tag{8.12}$$

From the definition in equation 8.6 it follows that

$$\lim_{z\to 1} P^{(1)}(z) \;=\; \lim_{z\to 1} \sum_{k=1}^{\infty} k\, p_k\, z^{k-1} \;=\; \sum_{k=1}^{\infty} k\, p_k \;=\; \overline{N}(1), \tag{8.13}$$

where $P^{(1)}(z)$ represents the first derivative of $P(z)$ with respect to z, and $\overline{N}(1)$ is the average number of tokens in the single β-unit system including the token currently being processed.

For g_i we verify that

$$\lim_{z\to 1} G^{(2)}(z) \;=\; \lim_{z\to 1} \sum_{i=0}^{\infty} i\,(i-1)\, g_i\, z^{i-2} \;=\; \sum_{i=1}^{\infty} i\,(i-1)\, g_i \;=$$
$$\sum_{i=1}^{\infty} i^2\, g_i \;-\; \sum_{i=1}^{\infty} i\, g_i \;=\; \overline{G^2} \;-\; \overline{G}, \tag{8.14}$$

where $G^{(2)}(z)$ is the second derivative of $G(z)$ with respect to z, and $\overline{G^2}$ is called the *second moment* of $G(z)$. This indicates that the limit as z goes to 1 of the second derivative of $G(z)$ is equal the difference between the second and the first moments of $G(z)$.

In a similar fashion, we can establish that

$$\lim_{z\to 1} H^{(2)}(z) \;=\; \overline{H^2} \;-\; \overline{H}. \tag{8.15}$$

Calculating the derivative of expression 8.5, taking the limit as z goes to 1 (see Appendix C), and using the results 8.10, 8.11, 8.13, 8.14, and 8.15, we obtain the following expression for $\overline{N}(1)$ in terms of the moments of $G(z)$ and $H(z)$.

$$\overline{N}(1) = \frac{\rho_1}{2(1-\rho_1)}\left[\frac{(\overline{H^2}+\overline{H}}{\overline{H}} + \frac{(\overline{G^2}-\overline{G}}{1-\overline{G}}\right] \tag{8.16}$$

Since the values that appear on the right hand side of equation 8.16 can be measured in the simulator described in Chapter 6, we can use this equation to estimate the average number of tokens in the single β-unit Rete Network. We use Little's result to obtain the average time that a token spends in the system [54, 45].

$$\overline{T}(1) = \frac{\overline{N}(1)(1-\overline{G})}{\lambda_\beta \overline{H}}, \tag{8.17}$$

where $\overline{T}(1)$ represents the average total time that a token spends in a single β-unit system, including the time the token is waiting in the queue and the processing time. Observe that we have to consider the total arrival rate in the β-FIFO, i.e., external tokens originated in the α-nodes as well as tokens generated in the β-nodes. In equation 8.17, the external arrival rate is $\lambda_\beta \overline{H}$ and the arrival rate due to generation of tokens in β-units is $(1-\overline{G})^{-1}$.

8.3.2 Multiple β-Unit Model

In this section we are interested in estimating the average time that a token spends in the m β-units organization shown in Figure 8.2.

Figure 8.4 presents the state-transition-rate diagram for the system with m β-units. The main difference from the single β-unit model is that the transition rate out of states 1 through $m-1$ due to the processing of tokens in β-units is proportional to the number of tokens in the system. This occurs because if $i < m$ tokens are present, all tokens are being processed, and $m-i$ β-units

Figure 8.4: State Diagram for System with m β-Units

The m–1 boundary conditions of this system are expressed by equation 8.18. The flow conservation equation for state $k \geq m$ is given by equation 8.19.

For $k < m$:

$$(\lambda_\beta + k\,\mu_\beta)\,p_k = (k+1)\,\mu_\beta\,g_0\,p_{k+1} + \lambda_\beta \sum_{j=0}^{k-1} p_j\,h_j + \mu_\beta \sum_{i=1}^{k} i\,p_i\,g_{k-i+1}. \qquad (8.18)$$

For $k \geq m$:

$$(\lambda_\beta + m\,\mu_\beta)\,p_k = m\,\mu_\beta\,g_0\,p_{k+1} + \lambda_\beta \sum_{l=0}^{k-1} p_l\,h_l + \mu_\beta \sum_{i=1}^{m-1} i\,p_i\,g_{k-i+1} + \mu_\beta \sum_{j=m}^{k} m\,p_j\,g_{k-j+1}. \tag{8.19}$$

To compute the generating function $P(z)$, we have to sum equation 8.18 from 0 to $m-1$, and sum equation 8.19 from m to infinity, and then combine both results and simplify (see appendix C). This results in

$$P(z) = \frac{\mu_\beta\,[z - G(z)]\,\sum_{k=0}^{m-1}(m-k)\,p_k\,z^k}{\lambda_\beta\,z\,[1 - H(z)] + m\,\mu_\beta\,[z - G(z)]}. \tag{8.20}$$

Computing the limit of $P(z)$ as z goes to 1, and making the result equal to 1 (series sum property), we obtain relation 8.21. Equation 8.18 provides m equations and $m+1$ unknowns. The relation expressed in 8.21 is the last equation we need to solve this linear system and obtain the values for the boundary probabilities p_0, p_1, ..., p_m.

$$\sum_{k=0}^{m-1}(m-k)\,p_k = m\,(1 - \rho_m) \tag{8.21}$$

$$\rho_m = \frac{\lambda_\beta\,\overline{H}}{m\,\mu_\beta\,(1 - \overline{G})} \tag{8.22}$$

To guarantee stability, the utilization factor ρ_m for the system with m β-units must be less than unity. Therefore the distributions of g_i and h_i must have the following property:

$$\frac{\overline{H}}{1 - \overline{G}} < \frac{\lambda_\beta}{m\,\mu_\beta}. \tag{8.23}$$

The average number of tokens in the system with m β-units, including the tokens that are currently being processed, is obtained by computing the limit of the first derivative of $P(z)$ as z goes to 1 (see appendix C).

$$\overline{N}(m) = \frac{\rho_m}{2(1-\rho_m)}\left[\frac{(\overline{H^2}+\overline{H})}{\overline{H}} + \frac{(\overline{G^2}-\overline{G})}{1-\overline{G}}\right] + \frac{\sum_{k=0}^{m-1} k\,(m-k)\,p_k}{m\,(1-\rho_m)}. \quad (8.24)$$

Note that expression 8.24 is identical to expression 8.16 when $m = 1$, as expected. The first and second moments of $G(z)$ and $H(z)$ are obtained from measures of g_i and h_j in the simulator. The boundary probabilities are obtained by solving the system of linear equations mentioned previously.

8.3.3 Rete Processing Improvement

Using Little's result, the ratio between the average time spent in the single β-unit system and the average time spent in the system with m β-units can be obtained from the average number of tokens in each one of these systems. We define the improvement in system time according to equation 8.25.

$$I_s(m) = \frac{\overline{T}(1)}{\overline{T}(m)} = \frac{\overline{N}(1)}{\overline{N}(m)}, \quad (8.25)$$

assuming that the effective arrival rate of the system does not change with the number of β-units. $I_s(m)$ indicates how much faster a token goes through the β-node portion of the Rete Network when m β-units are used instead of one.

We purposely avoid calling this performance improvement *speedup* because of the loaded meaning of that word. Speedup is a system level concept that is applied to different settings, e.g. fixed-time speedup, fixed-load speedup, memory-bound speedup, etc [37]. Also, except for very rare cases, the speedup of a computer system is always a sublinear function of the number of processors

in the system. In our system, because of the nature of the incoming flow of tokens, the amount of improvement in the time spent in the β-network can be superlinear. This happens because the existence of occasional bursts of traffic in the incoming flow of tokens can cause tokens to spend considerable amount of time waiting in the β-unit input queue. Of course, if the incoming flow of token were to be steady, the improvement in the time spent in the system would be at most linear.

8.4 Experimental Results

The analytical model presented in this chapter assumes a steady state in the flow of tokens through the Rete Network. An approximation of such a situation is only encountered in fairly large production system programs. We will present measurements for the benchmark `moun2`, which is the largest benchmark presented in Chapter 4. Table 8.1 present measurements for the first and second moment of g_i and h_j, for the service rate μ_β and the average number of tokens in the single β-node system $\overline{N}(1)$ as measured in the event driven simulator described in Chapter 6.

# Proc	$\overline{G}$	$\overline{G^2}$	$\overline{H}$	$\overline{H^2}$	μ_β	$\overline{N}(1)$
1	0.78	3.35	5.45	40.22	0.00096	644.8
2	0.80	3.10	2.89	11.58	0.00093	244.4
4	0.79	4.17	2.15	6.15	0.00096	160.2
6	0.80	4.20	1.59	3.51	0.00094	89.6
10	0.80	5.06	1.41	2.48	0.00102	47.8

Table 8.1: Parameter Measurements for `moun2`.

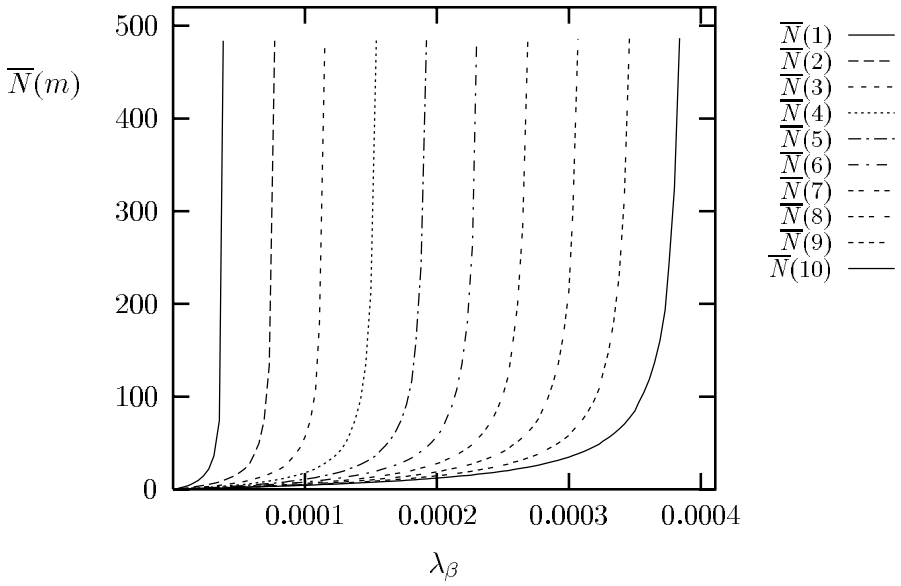

Figure 8.5: Average Number of Tokens in the System versus λ_β for `moun2` in a Single Processor Architecture

In the architecture presented in Chapter 3, the production set is divided among the processors in the machine. Therefore, a machine with a larger number of processors has smaller Rete Networks in each processor. Moreover, the amount of tokens processed in each Rete Network is smaller. Note that the first and second moments of g_i do not change significantly when the Rete Network is divided into a larger number of networks, but rather seems to be a characteristic of the benchmark program. On the other hand, the moments of h_j do change substantially when each processor has smaller Rete Networks because in such networks each α-node has fewer successors. The value of μ_β is also independent of the size of the Rete Network. The measures presented in Table 8.1 were obtained by forcing the simulator to implement a single β-FIFO to replicate the

simplification used in the analytical model.

Figure 8.5 shows the variation in the average number of tokens in the system with the value of the arrival rate λ_β for Rete Network of *moun*2, in a single processor architecture with one up to ten β-units. Notice that there is a dramatic increase in the number of tokens in the system when the utilization rate tends to one. For systems operating close to such an asymptote, the addition of a single β-unit can improve the performance significantly. This improvement is due to the reduction in the amount of time spent waiting in the system.

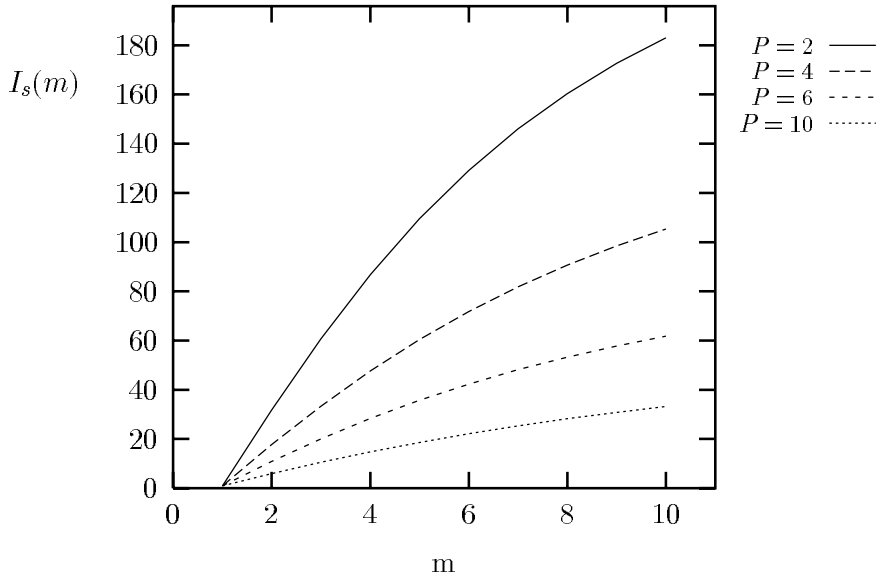

Figure 8.6: Speed improvement prediction for `moun2` considering λ_β constant for all values for m.

A difficulty in the utilization of this analytical model to estimate performance improvements is the correct estimation of arrival rate λ_β. We can obtain the arrival rate for a single β-unit system from direct measurement in the simulator.

However, this rate does not remains constant when the number of β-units is increased. The Instantiation Firig Engine works as an outer loop that receives new instantiations from the output of the Rete Network and generates new actions that are placed in the Rete Network input. A Rete Network with a larger number of β-units produces changes to the conflict set at a faster pace resulting in a higher arrival rate. Figure 8.6 presents the estimated speed improvement for machines with various numbers of processors considering that the arrival rate remains constant when the number of β-units is changed.

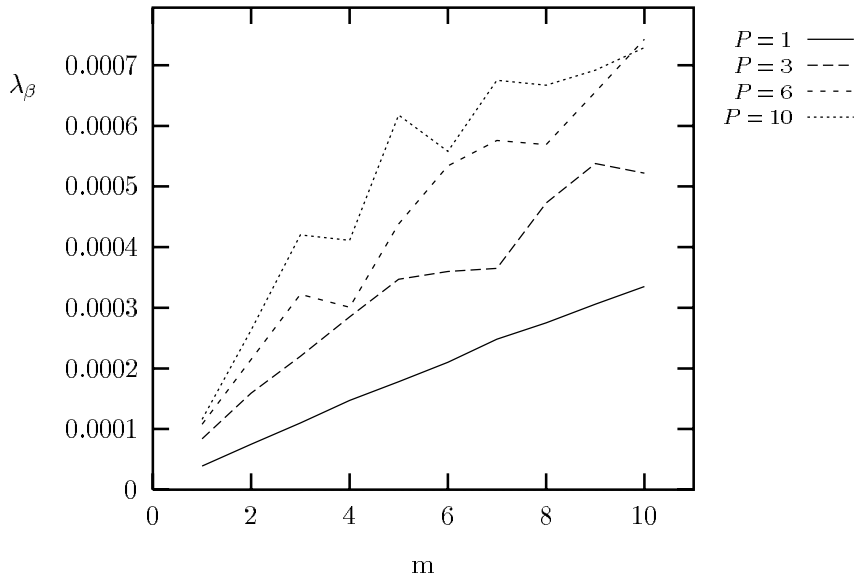

Figure 8.7: Variation of λ_β with m for `moun2` (P indicates the number of processors in the architecture).

Because the simulator described in Chapter 6 has the capacity to simulate multiple β-units Rete Network, we can measure the amount of change in the arrival rate λ_β when the number of β-units is increased. The variation of λ_β

with the number of β-units in the Rete Network is shown in Figure 8.7. The curves in this graph indicate that the increasing in the arrival rate λ_β might be approximated by linear functions.

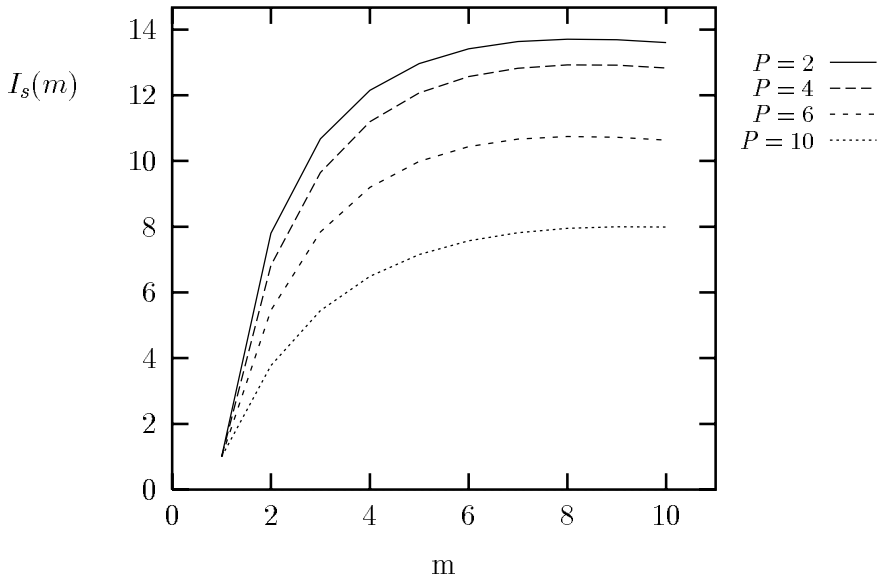

Figure 8.8: Speed improvement prediction for `moun2` considering that λ_β changes when more β-units are used.

Using this measurements, we obtain the speedup curves shown in Figure 8.8. Observe the significant reduction in speedup when these curves are compared with the curves in Figure 8.6. Future research with experimental measurements of a larger set of benchmarks might determine a general rule to estimate the variation of the arrival rate with the number of β-units.

8.5 Final Remarks

This chapter presented a complete analytical model to study the performance of multifunctional unit Rete Networks. The results show that in certain configurations, a modest number of extra β-units might eliminate wasted waiting time, producing significant improvement in the average time taken for a token to be processed in the Rete Network. Further studies along this line of research include the following: construction of an analytical model for the priority system with an external and an internal β-queue; an analytical model for the outer loop formed by the instantiation firing engine and the broadcasting network, which takes into consideration the interactions between all the Rete Networks located in the different processors in the machine; and further experimental studies with this model to determine how the arrival rate changes when the number of β-units is increased.

Chapter 9

Conclusion

This dissertation presented a new architecture for parallel production systems. The development of a comprehensive event-driven simulator allowed various performance measurement for this architecture. The results demonstrate that the adoption of serializability as a correctness criterion might lead to significant performance improvements. Such improvements are obtained through the elimination of global synchronization and through the overlapping between different phases of the production system execution. The use of modern associative storage techniques was key in the design of the architecture.

The development of the Contemporaneous Traveling Salesperson Problem was not only useful for our measurements, but may also be useful for many researchers looking for versatile benchmarking facilities in the future.

The history of research in production systems indicates that architectures with thousands of processors tend to deliver poor price/performance ratios. The proposed multiple functional unit Rete Network utilizes a modest number

of functional units and produces considerable speedup over a single processor architecture.

In spite of being a powerful tool, analytic modeling has not been used much in the study of performance of Production Systems. This shortcoming was addressed through the development of an analytical model for the study of performance of the multiple functional unit Rete Network.

Future research might be conducted on a number of issues. The architecture proposed in this dissertation executes a compiled Production System. Therefore, compiling optimization techniques such as the ones proposed by Kuo, Miranker and Browne [49] can be incorporated in the compiler.

Acharya and Tambe [2] have reported considerable speedup in the processing of Production Systems that manipulate collection of WMEs instead of one WME at a time. An interesting area of research would be the study of the execution of collection-oriented Production Systems in the architecture proposed. One problem still to be overcome in such systems is a proper handling of self disabling productions. A smaller step that would deliver only limited speedup would be to change the architecture to allow propagation of modify tokens through the Rete Network.

The use of serializability as a correctness criterion imposes a heavier burden on the programmer to assure that a program is correct. Our experience with PS benchmarks indicates that programmers often rely on knowledge about conflict set resolution strategies when writing PS programs. This is mostly evidenced by the omission of important antecedents in productions that are enabled but never selected to fire by an specific strategy. For problems like CTSP, writing a serializable correct PS was fairly straightforward. Now that our study has indicated that serializable systems offer great speed improvements, it is desirable

to develop programming aid tools to help in the specification and verification of a wider range of serializable PS programs.

Improvements to the analytical model include the development of a model for the operation of the Instantiation Firing Engine and the interaction among processors through the Broadcast Interconnection Network. An alternative to this would be the conduction of an empirical study to determine the relationship between the number of functional units in the Rete Network and the ratio of external arrivals in the α-nodes. Another improvement to the analytical model would be the extension of the model developed to consider a priority system with the external and internal β-queues.

Appendix A

Performance Measurements

This appendix presents the results of an extensive empirical study with the simulator for most of the benchmarks described in Chapter 4. The architecture simulated is the one described in Chapter 3 with the multiple β-unit Rete Network organization introduced in Chapter 7. The plots presented in the following pages were interpolated to better show the behavior of the machine as it is augmented with more processors and β-functional units.

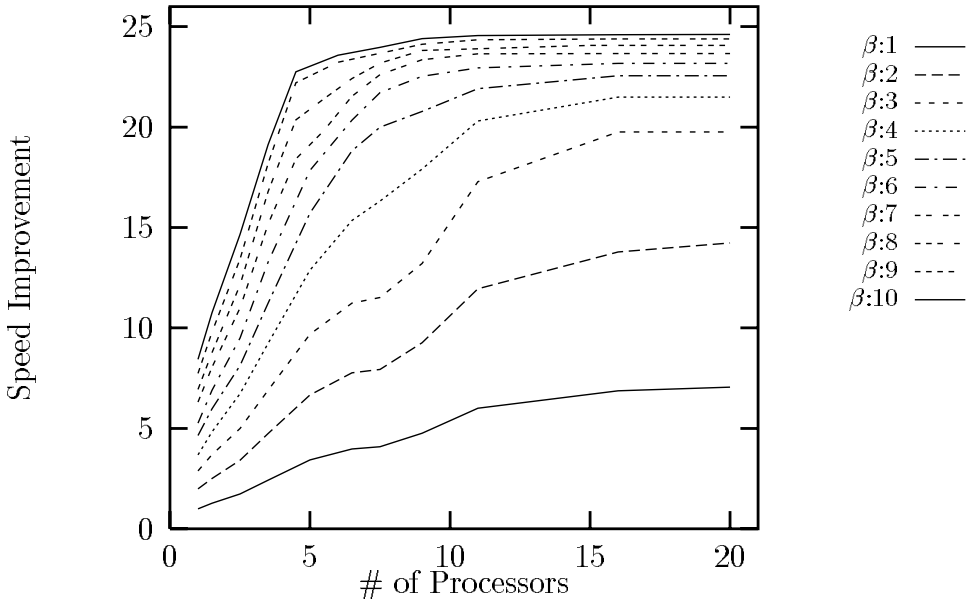

Figure A.1: Speedup Curves for `patents`

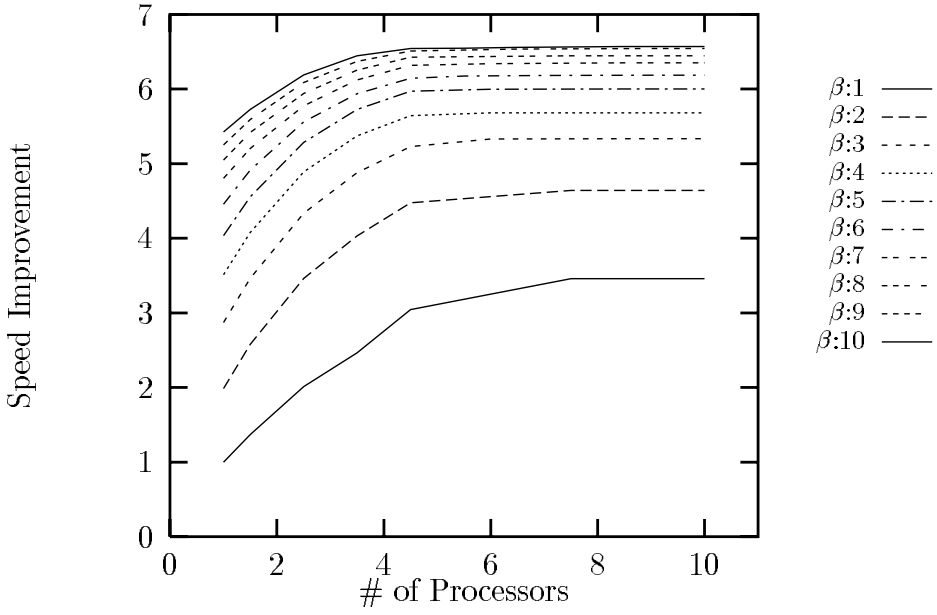

Figure A.2: Speedup Curves for `waltz2`

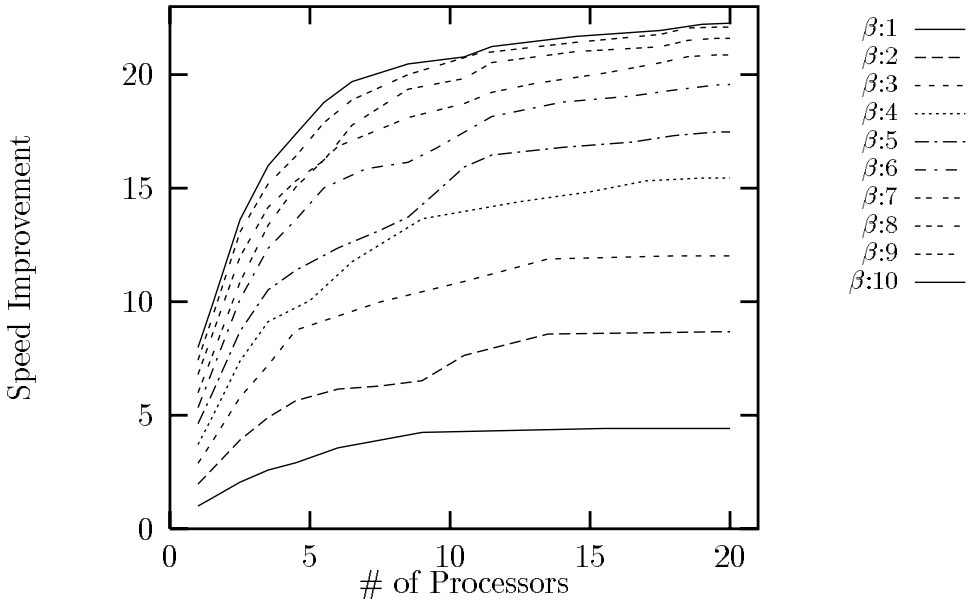

Figure A.3: Speedup Curves for `south`

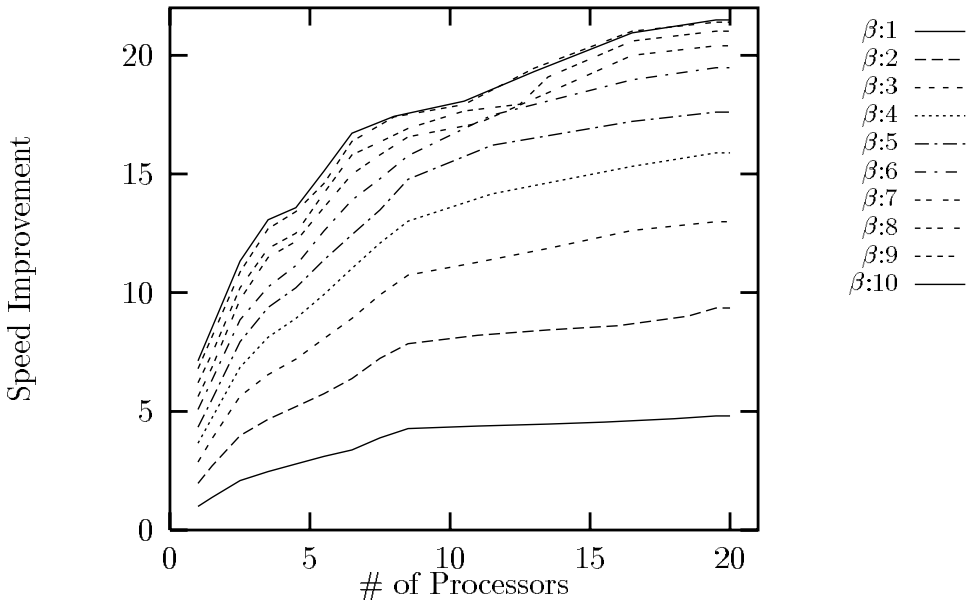

Figure A.4: Speedup Curves for `south2`

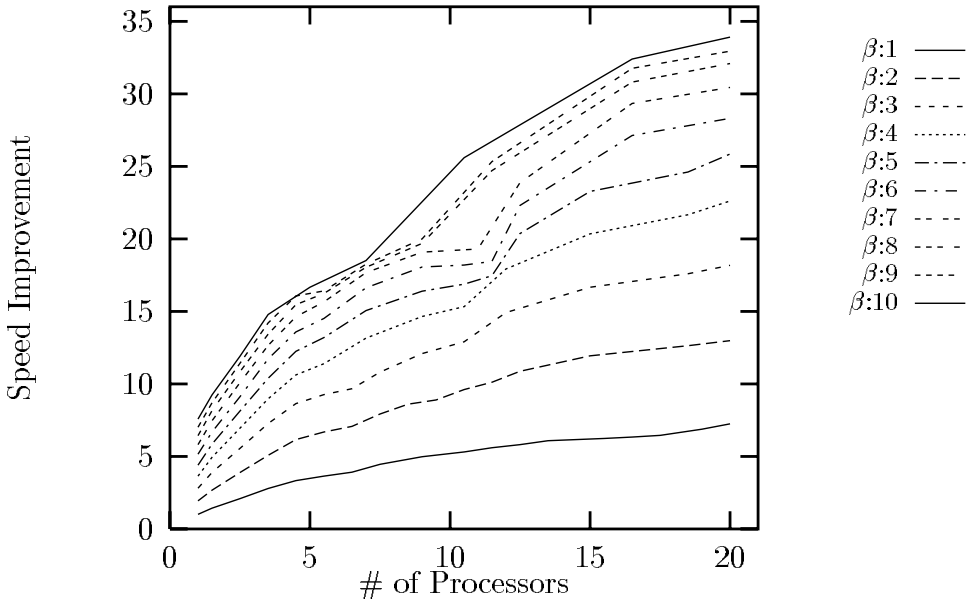

Figure A.5: Speedup Curves for `moun2`

Appendix B

Study of CTSP Benchmark

This appendix presents an extensive study of the architecture performance with the CSTP benchmark presented in chapter 4. All experiments were performed using an instantiation of CTSP with seven "states"[1].

The map with the states is shown in Figure B.1. We performed two set of experiments. First, we maintain the standard deviation σ_c constant and change the average number of cities in each state in order to study the effect of problem size on performance. Then we maintain the average number of cities in each state constant and change the standard deviation. This allows us to study the effect of load inbalance among the processors. Because the number of states is small, the actual mean and standard deviation might not be exactly what was specified in the benchmark generator. Table B.1 presents the specified and the measured mean and standard deviation for each benchmark studied. It also

[1] In the description of CTSP in Chapter 4 a group of cities form a "country" and a group of countries form a "continent". In the implementation used for the experiments presented in this appendix, cities form states and states form countries.

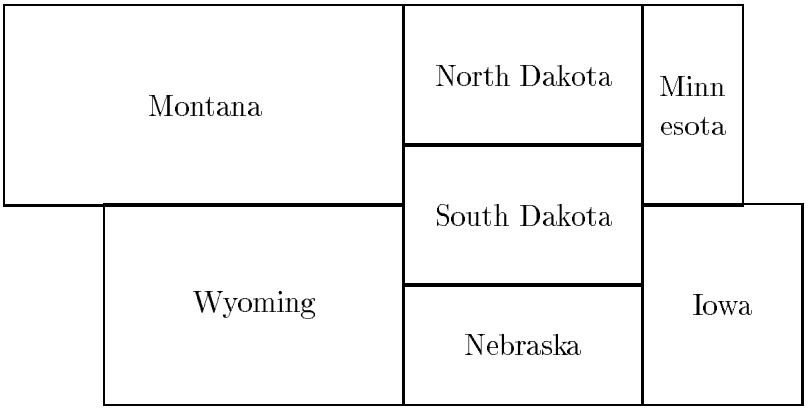

Figure B.1: Country map.

shows the total number of cities in each instantiation.

Benchmark	Specified		Measured		Total #
	μ_c	σ_c	μ_c	σ_c	of Cities
d_c06_v3	6	3	8.8	3.4	58
d_c10_v3	10	3	11.1	2.1	78
d_c20_v3	20	3	20.9	2.8	122
d_c30_v3	30	3	29.4	1.7	206
d_c15_v0	15	0	15.0	0.0	105
d_c15_v3	15	3	16.0	2.2	112
d_c15_v6	15	6	15.6	5.8	109
d_c15_v9	15	9	14.6	7.8	102

Table B.1: Specified and actual values for μ_c and σ_c for the benchmarks studied in this appendix.

Table B.2 shows the actual distribution of the number of cities for each state in the map of Figure B.1.

Figures B.2 and B.3 plot results for the first experiment in which the mean

Bench.	Mont.	N.Dak.	S.Dak.	Wyoming	Minnes	Nebr.	Iowa
d_c06_v3	6	15	6	6	7	6	12
d_c10_v3	14	10	8	14	10	12	10
d_c20_v3	22	21	20	20	20	20	19
d_c30_v3	28	31	30	30	31	26	30
d_c15_v0	15	15	15	15	15	15	15
d_c15_v3	20	16	15	13	14	16	18
d_c15_v6	9	18	21	18	24	8	11
d_c15_v9	21	21	16	8	9	2	25

Table B.2: Distribution of number of cities per state.

number of cities per state is changed while its variance remains constant. Figure B.2 presents results for an architecture with a single β-unit Rete Network. Figure B.3 shows the speed improvement as the number of processors is increased in a machine with a 10 β-unit Rete Network. The base of comparison for the curves of figure B.3 is a machine with a single processor and 10 β-units in the Rete Network.

Figures B.4 and B.5 present the results for the experiment that changes the standard deviation for the number of cities in each state while maintaining its mean constant. Figure B.4 plots the curves for a single β-unit architecture while Figure B.5 presents the results for a 10 β-unit architecture. The base of comparison for the curves in Figure B.5 is a single processor architecture with 10 β-units.

Figure B.6 plots the amount of speedup obtained for an instantiation of CTSP with seven states and $\mu_c = 15$ when the standard deviation for the number of cities σ_c is changed. The amount of speedup obtained decreases

when there is a higher variance in the number of cities in each state. This is an expected effect because higher variance in the size of local clusters causes more inbalance in the workload of different processors.

Figure B.7 plots the number of cycles performed to solve instantiations of CTSP in function of the average number of cities per state μ_c. Figure B.7 shows curves for architectures with 1, 8, and 16 processors. Although the number of cycles still grows exponentially with the number of cities in the parallel architecture, the processing time is significantly reduced allowing the parallel architecture to tackle larger problems.

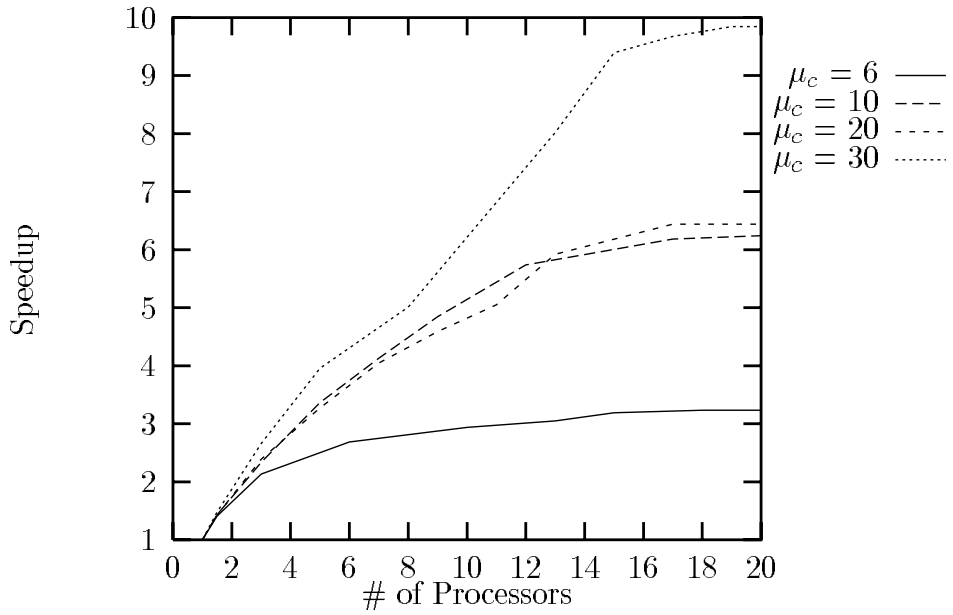

Figure B.2: Speedup Curves for single β-unit architecture ($\sigma_c = 3$).

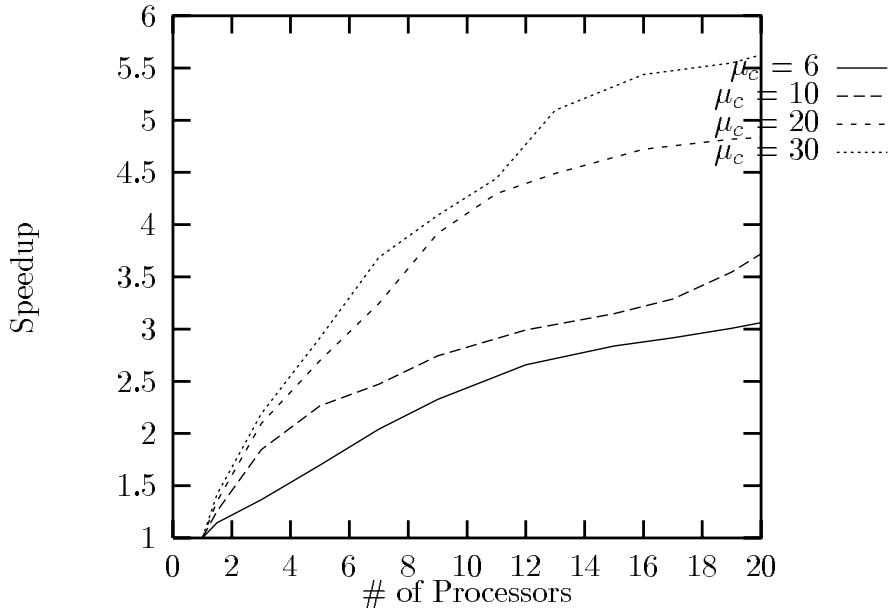

Figure B.3: Speedup Curves for 10 β-unit architecture ($\sigma_c = 3$).

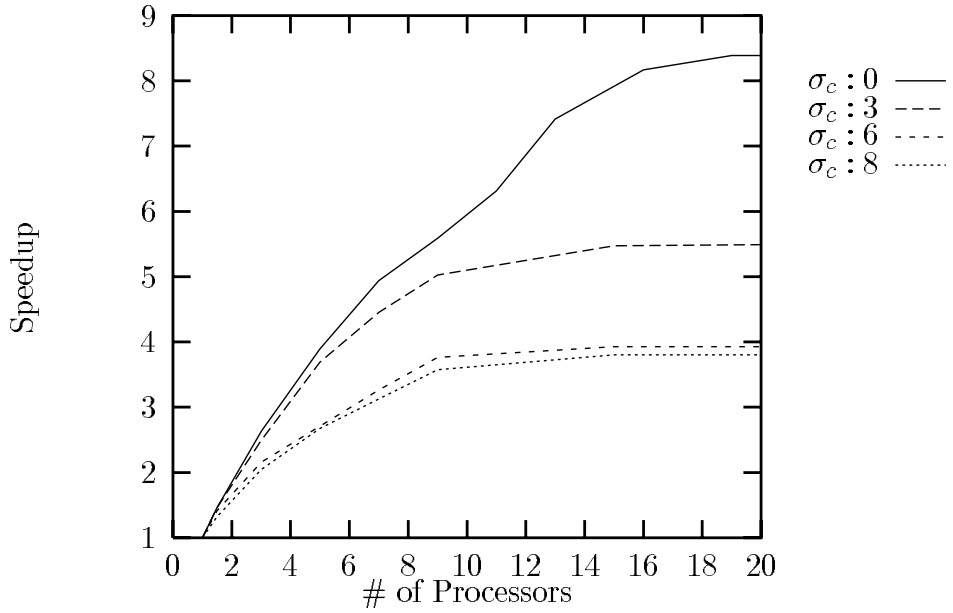

Figure B.4: Speedup Curves for single β-unit architecture($\mu_c = 15$).

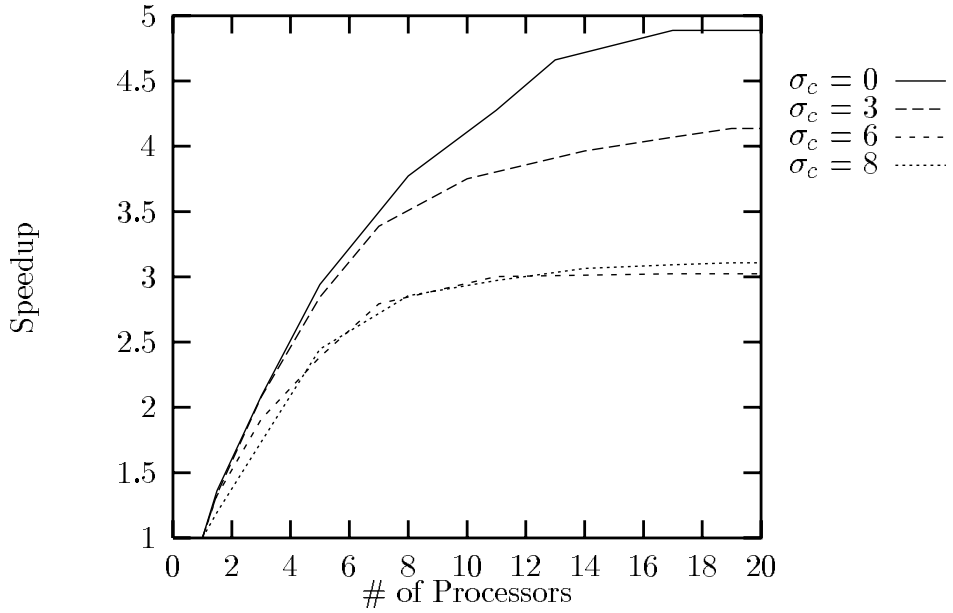

Figure B.5: Speedup Curves for 10 β-unit architecture ($\mu_c = 15$).

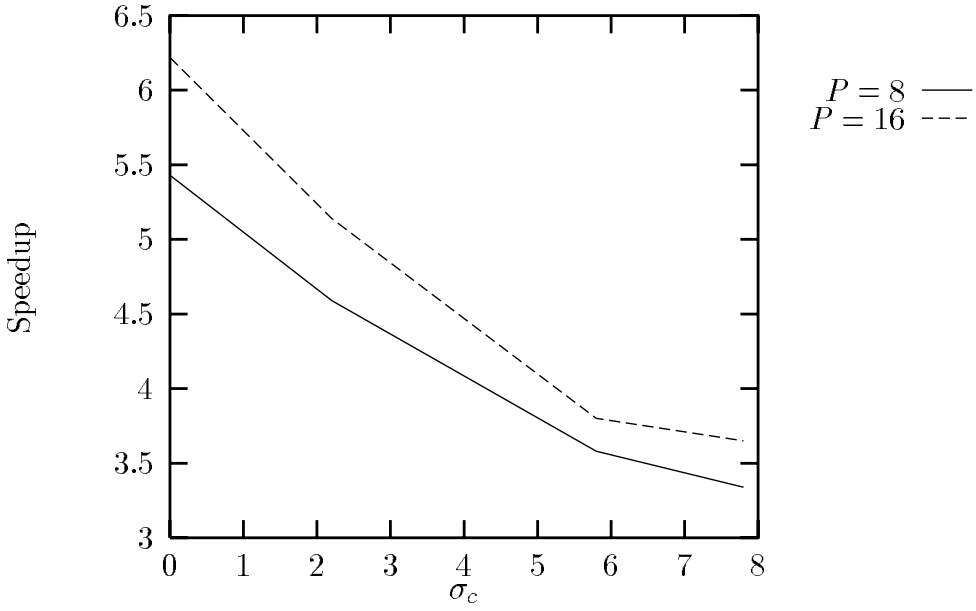

Figure B.6: Speedup for single β-unit architecture versus the standard deviation in the number of cities σ_c ($\mu_c = 15$).

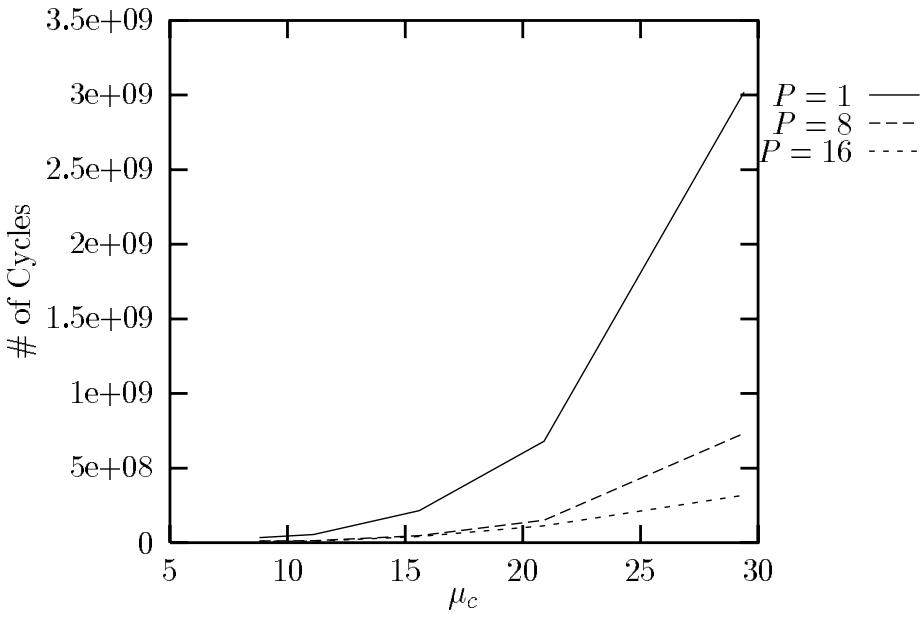

Figure B.7: Number of computing cycles in the single β-unit architecture versus average number of cities per state ($\sigma_c = 3$).

Appendix C

Derivation of Analytical Model Expressions

C.1 Single β-Unit Model

In this section we present a detailed derivation of equation 8.5 for the probability generating function $P(z)$, and for equation 8.16 of chapter 8 that allows the computation of the average number of tokens in the system from the first and second moments of $G(z)$ and $H(z)$, for single β-unit system.

C.1.1 Generating Function for Single β-Unit System

We reproduce here the steady-state equations 8.1 and 8.4 obtained through the conservation of flow principle.

$$(\lambda_\beta + \mu_\beta)\, p_k = \mu_\beta\, g_0\, p_{k+1} + \lambda_\beta \sum_{j=0}^{k-1} p_j\, h_{k-j} + \mu_\beta \sum_{i=1}^{k} p_i\, g_{k-i+1}, \quad \text{(C.1)}$$

$$\lambda_\beta\, p_0 = \mu_\beta\, g_0\, p_1 \quad \text{(C.2)}$$

where p_k is the probability of k tokens being in the system, including the token currently being processed, h_j is the probability that an external bulk arrival has j tokens, g_i is the probability that the processing of a token produces i new tokens, λ_β is the external arrival rate into the β-queue, and μ_β is the processing rate on β-nodes.

We are interested in the generating function for the number of tokens in the system $P(z)$, expressed in terms of λ_β, μ_β, $H(z)$, and $G(z)$. First we multiply both sides of equation C.1 by z^k and then sum both sides from 1 to infinity, obtaining

$$\begin{aligned}(\lambda_\beta + \mu_\beta) \sum_{k=1}^{\infty} p_k z^k &= \mu_\beta g_0 \sum_{k=1}^{\infty} p_{k+1} z^k + \lambda_\beta \sum_{k=1}^{\infty} \sum_{j=0}^{k-1} p_j h_{k-j} z^k + \\ &\quad \mu_\beta \sum_{k=1}^{\infty} \sum_{i=1}^{k} p_i g_{k-i+1} z^k. \end{aligned} \tag{C.3}$$

The order of the indexes in the second double sum can be reversed as follows:

$$\begin{aligned}\sum_{k=1}^{\infty} \sum_{i=1}^{k} p_i g_{k-i+1} z^k &= \sum_{i=1}^{\infty} \sum_{k=i}^{\infty} p_i g_{k-i+1} z^k \\ &= \frac{1}{z} \sum_{i=1}^{\infty} p_i z^i \sum_{k=i}^{\infty} g_{k-i+1} z^{k-i+1}. \end{aligned} \tag{C.4}$$

Making the change of indexes $j = k - i + 1$ in the last sum, we obtain

$$\begin{aligned}\sum_{k=i}^{\infty} g_{k-i+1} z^{k-i+1} &= \sum_{j=1}^{\infty} g_j z^j = G(z) - g_0, \\ \sum_{k=1}^{\infty} \sum_{i=1}^{k} p_i g_{k-i+1} z^k &= \frac{1}{z}[P(z) - p_0][G(z) - g_0]. \end{aligned} \tag{C.5}$$

Similarly, for the first double sum, we obtain

$$\sum_{k=1}^{\infty} \sum_{j=0}^{k-1} p_j h_{k-j} z^k = P(z) H(z) \tag{C.6}$$

We also observe that

$$\sum_{k=1}^{\infty} p_k\, z^k \;=\; P(z) - p_0 \tag{C.7}$$

$$\sum_{k=1}^{\infty} p_{k+1}\, z^k \;=\; \frac{1}{z}\sum_{k=1}^{\infty} p_{k+1}\, z^{k+1} \;=\; \frac{1}{z}\sum_{j=2}^{\infty} p_j\, z^j$$

$$\;=\; \frac{1}{z}[P(z) \;-\; p_0 \;-\; z\, p_1] \tag{C.8}$$

$$\sum_{k=1}^{\infty} p_{k-1}\, z^k \;=\; z\sum_{k=1}^{\infty} p_{k-1}\, z^{k-1} \;=\; z\sum_{j=0}^{\infty} p_j\, z^j \;=\; z\, P(z) \tag{C.9}$$

Substituting C.5, C.6, C.7, C.8, and C.9 into C.3, we obtain

$$\begin{aligned}(\lambda_\beta \;+\; \mu_\beta)[P(z) \;-\; p_0] \;=\;& \mu_\beta\, g_0\, \frac{1}{z}[P(z) \;-\; p_0 \;-\; z\, p_1] \;+ \\ & \mu_\beta\, \frac{1}{z}[P(z) \;-\; p_0][G(z) - g_0] \;+ \\ & \lambda_\beta\, P(z)\, H(z)\end{aligned} \tag{C.10}$$

Using the boundary condition giving by equation C.2 into equation C.10 and simplifying, we obtain

$$P(z) = \frac{\mu_\beta\, p_0[z \;-\; G(z)]}{\lambda_\beta\, z\,[1 \;-\; H(z)] \;+\; \mu_\beta\,[z \;-\; G(z)]} \tag{C.11}$$

In order to obtain the value of p_0, we use the series sum property of the Z transform. From the definition of $P(z)$, we conclude that the limit of $P(z)$ as z goes to 1 is equal the sum of all probabilities p_k, and therefore must be equal to the unity.

$$\lim_{z\to 1} P(z) \;=\; \lim_{z\to 1}\sum_{k=0}^{\infty} p_k\, z^k \;=\; \sum_{k=0}^{\infty} p_k \;=\; 1 \tag{C.12}$$

In a similar fashion, both the limit of $G(z)$ and the limit of $H(z)$ as z goes to 1 must be equal to 1 because they represent the sum of all probabilities g_i and h_j respectively.

$$\lim_{z\to 1} G(z) \;=\; \lim_{z\to 1} \sum_{i=0}^{\infty} g_i\, z^i \;=\; \sum_{i=0}^{\infty} g_i \;=\; 1 \tag{C.13}$$

$$\lim_{z\to 1} H(z) \;=\; \lim_{z\to 1} \sum_{j=0}^{\infty} h_j\, z^j \;=\; \sum_{j=0}^{\infty} h_j \;=\; 1 \tag{C.14}$$

A first attempt to compute the limit of equation C.11 produces an indefinite result. By L'Hospital's rule,

$$\lim_{z\to 1} P(z) \;=\; \frac{\mu_\beta\, p_0\,[1 \;-\; \lim_{z\to 1} G^{(1)}(z)]}{\mu_\beta\,[1 \;-\; \lim_{z\to 1} G^{(1)}(z)] \;-\; \lambda_\beta \lim_{z\to 1} H^{(1)}(z)}, \tag{C.15}$$

where $G^{(1)}(z)$ is the first derivative of $G(z)$ and $H^{(1)}(z)$ is the first derivative of $H(z)$. We observe that

$$\lim_{z\to 1} G^{(1)}(z) \;=\; \sum_{i=0}^{\infty} i\, g_i \;=\; \overline{G} \tag{C.16}$$

$$\lim_{z\to 1} H^{(1)}(z) \;=\; \sum_{j=0}^{\infty} j\, h_j \;=\; \overline{H} \tag{C.17}$$

$$, \tag{C.18}$$

where $\overline{G}$ and $\overline{H}$ are the first moments of $G(z)$ and $H(z)$, respectively. Replacing $\overline{G}$ and $\overline{H}$ in equation C.15 and using the result of equation C.12 we obtain

$$p_0 \;=\; 1 \;-\rho_1, \tag{C.19}$$

$$\rho_1 \;=\; \frac{\lambda_\beta\, \overline{H}}{\mu_\beta\,(1-\overline{G})}. \tag{C.20}$$

Substituting C.19 in C.11 results

$$P(z) = \frac{\mu_\beta\,(1 \;-\; \rho_1)[z \;-\; G(z)]}{\lambda_\beta\, z\,[1 \;-\; H(z)] \;+\; \mu_\beta\,[z \;-\; G(z)]}. \tag{C.21}$$

Equation C.21 is the desired probability generating function for the number of tokens in the system(Eq. 8.5).

To verify expression C.21 we might consider special cases. If there is no "feedback" from the β-units to the β-FIFO, $g_0 = 1$ and $g_i = 0$ for $i \neq 0$. In this case $G(z) = 1$ and expression C.21 becomes

$$P(z) = \frac{\mu_\beta\,(1\ -\ \rho_1)(z-1)}{\lambda_\beta\, z\,[1\ -\ H(z)]\ +\ \mu_\beta\,(z-1)}. \tag{C.22}$$

that is exactly the generating probability function for an M/M/1 system with bulk arrival [45].

If all the bulks arriving from α-units have a single token, i.e., $h_1 = 1$ and $h_i = 0$ for $i \neq 1$, $H(z) = z$, expression C.22 becomes

$$P(z) = \frac{1-\rho_1}{1-\rho_1\, z} \tag{C.23}$$

which is the generating probability function for an M/M/1 system.

C.1.2 Average Number of Tokens in a Single β-Unit System

In this section we present a detailed derivation of equation 8.16 that computes the average number of tokens in a single β-unit $\overline{N}(1)$. According to equation 8.13, $\overline{N}(1)$ is given by the limit of the first derivative of $P(z)$ as z approximates the unity.

To facilitate the computation of this limit, we rewrite equation C.21 as a quotient of two polynomial functions in z.

$$P(z) = K\,\frac{N(z)}{D(z)} \tag{C.24}$$

For the single β-unit system that we are studying in this section, we have

$$K \quad = \quad \mu_\beta\,(1\ -\ \rho_1) \tag{C.25}$$

$$N(z) = z - G(z) \tag{C.26}$$

$$D(z) = \lambda_\beta z[1 - H(z)] + \mu_\beta [z - G(z)] \tag{C.27}$$

From C.24 we obtain the derivative of $P(z)$

$$P'(z) = K \frac{N'(z)\,D(z) - N(z)\,D'(z)}{[D(z)]^2}. \tag{C.28}$$

Observe that for the generating function $P(z)$ expressed by equation C.21, both the limit of the numerator and the limit of the denominator are equal to zero.

$$\lim_{z\to 1} N(z) = \lim_{z\to 1} D(z) = 0. \tag{C.29}$$

The direct computation of the limit of equation C.28 produces an indefinite result. Using L'Hospital's rule, we obtain

$$\lim_{z\to 1} P'(z) = K \lim_{z\to 1} \frac{N''(z)\,D(z) - N(z)\,D''(z)}{2\,D(z)\,D'(z)}. \tag{C.30}$$

Because of the condition expressed in C.29 the computation of the limit in C.30 still results in an indefinite result. The application of L'Hospital's rule a second time, results in

$$\begin{aligned} \lim_{z\to 1} P'(z) = {} & K \lim_{z\to 1} \frac{N'''(z)\,D(z) + N''(z)\,D'(z)}{2\,[D'(z)\,D'(z) + D(z)\,D''(z)]} - \\ & K \lim_{z\to 1} \frac{N'(z)\,D''(z) + N(z)\,D'''(z)}{2\,[D'(z)\,D'(z) + D(z)\,D''(z)]}. \end{aligned} \tag{C.31}$$

But we know from equation C.29 that all the non-differentiated expressions have limit equal to zero, therefore we can simplify C.31

$$\lim_{z\to 1} P'(z) = K \lim_{z\to 1} \frac{N''(z)\,D'(z) - N'(z)\,D''(z)}{2\,[D'(z)]^2} \tag{C.32}$$

When the expressions for the derivatives are complex, it might be easier to rewrite equation C.32 as

$$\lim_{z\to 1} P'(z) = \frac{K}{2}\left[\lim_{z\to 1}\frac{N''(z)}{2\,D'(z)} - \lim_{z\to 1}\frac{N'(z)\,D''(z)}{2\,[D'(z)]^2}\right] \tag{C.33}$$

It is important to observe that this result is independent of the generating function and is true as long as $P(z)$ can be expressed as a quotient of polynomial functions and the conditions expressed by equation C.29 hold. We will use expressions C.32 and C.33 on the derivation of the number of tokens in the m β-unit system later in this appendix.

Recalling the expressions for the limit of the first and second derivatives of $G(z)$ and $H(z)$ given in equations 8.10, 8.11, 8.14, and 8.15, we can compute the limits of the derivatives of $N(z)$ and $D(z)$:

$$\lim_{z\to 1} N'(z) = 1-\overline{G}, \tag{C.34}$$

$$\lim_{z\to 1} N''(z) = -(\overline{G^2}-\overline{G}), \tag{C.35}$$

$$\lim_{z\to 1} D'(z) = \mu_\beta\,(1-\overline{G})\,(1-\rho_1), \tag{C.36}$$

$$\lim_{z\to 1} D''(z) = -\lambda_\beta\,(\overline{H^2}+\overline{H}) - \mu_\beta\,(\overline{G^2}-\overline{G}). \tag{C.37}$$

Substituting equations C.25, C.34, C.35, C.36, and C.37 into equation C.33, we obtain

$$\overline{N}(1) = \frac{-(\overline{G}-\overline{G^2})}{2\,(1-\overline{G})} + \frac{[\lambda_\beta(\overline{H^2}-\overline{H}) + \mu_\beta(\overline{G^2}-\overline{G})]}{2\,\mu_\beta\,(1-\overline{G})\,(1-\rho_1)} \tag{C.38}$$

After some simplifications, equation C.38 yields

$$\overline{N}(1) = \frac{\rho_1}{2(1-\rho_1)}\left[\frac{(\overline{H^2}+\overline{H})}{\overline{H}} + \frac{(\overline{G^2}-\overline{G})}{1-\overline{G}}\right] \quad \text{(C.39)}$$

Equation C.39 is the desired expression (equation 8.16) for the average number of customers in the single β-unit.

C.2 Multiple β-Unit Model

The derivation of the generating function for the number of tokens in the system with m β-units is a bit more involved than the one for the single functional unit system. This is due to the existence of m boundary equations in this system. These equations appear because whenever some β-units are idle, the processing rate is proportional to the number of tokens in the system.

C.2.1 Generating Function for Multiple β-Unit System

In order to facilitate the manipulation of the equations, we introduce the following notation due to Chang and Harn [11].

$$P_i(z) = \sum_{j=0}^{i} p_j z^j, \qquad P(z) = P_\infty(z), \qquad P_i'(z) = \sum_{j=1}^{i} j\, p_j\, z^{j-1} \quad \text{(C.40)}$$

$$G_i(z) = \sum_{j=0}^{i} g_j z^j, \qquad G(z) = G_\infty(z), \qquad G_i'(z) = \sum_{j=1}^{i} j\, g_j\, z^{j-1} \quad \text{(C.41)}$$

$$H_i(z) = \sum_{j=0}^{i} h_j z^j, \qquad H(z) = H_\infty(z), \qquad H_i'(z) = \sum_{j=1}^{i} j\, h_j\, z^{j-1} \quad \text{(C.42)}$$

We reproduce here equation 8.19 for flow conservation for $k \geq m$

$$(\lambda_\beta + m\,\mu_\beta)\, p_k = m\,\mu_\beta\, g_0\, p_{k+1} + \lambda_\beta \sum_{l=0}^{k-1} p_l\, h_{k-l} +$$

$$\mu_\beta \sum_{i=1}^{m-1} i\, p_i\, g_{k-i+1} \; + \;$$
$$\mu_\beta \sum_{j=m}^{k} m\, p_j\, g_{k-j+1} \qquad \text{(C.43)}$$

We are interested in the generating function $P(z)$. We start by multiplying both sides of equation C.43 by z^k, and then we sum the equation from m to infinity.

$$\begin{aligned}(\lambda_\beta \; + \; m\,\mu_\beta) \sum_{k=m}^{\infty} p_k\, z^k \quad &= \quad m\,\mu_\beta\, g_0 \sum_{k=m}^{\infty} p_{k+1}\, z^k \; + \\ &\lambda_\beta \sum_{k=m}^{\infty} \sum_{l=0}^{k-1} p_l\, h_{k-l}\, z^k \; + \\ &\mu_\beta \sum_{k=m}^{\infty} \sum_{i=1}^{m-1} i\, p_i\, g_{k-i+1}\, z^k \; + \\ &\mu_\beta\, m \sum_{k=m}^{\infty} \sum_{j=m}^{k} p_j\, g_{k-j+1}\, z^k \qquad \text{(C.44)}\end{aligned}$$

Now we proceed to express each one of the terms in equation C.44 in terms of the notation defined in C.40, C.41, and C.42.

$$\begin{aligned}\sum_{k=m}^{\infty} p_k\, z^k \quad &= \quad \sum_{k=0}^{\infty} p_k\, z^k \; - \; \sum_{k=0}^{m-1} p_k\, z^k \\ &= \quad P(z) - P_{m-1}(z) \qquad \text{(C.45)} \\ \sum_{k=m}^{\infty} p_{k+1}\, z^k \quad &= \quad \frac{1}{z} \sum_{k=m}^{\infty} p_{k+1}\, z^{k+1} \\ &= \quad \frac{1}{z} \sum_{k=0}^{\infty} p_k\, z^k \; - \; \frac{1}{z} \sum_{k=0}^{m} p_k\, z^k \\ &= \quad \frac{1}{z}\,[P(z) - P_m(z)] \qquad \text{(C.46)} \\ \sum_{k=m}^{\infty} \sum_{l=0}^{k-1} p_l\, h_{k-l}\, z^k \quad &= \quad \sum_{l=0}^{m-1} p_l\, z^l \sum_{k=m}^{\infty} h_{k-l}\, z^{k-l} \; + \; \sum_{l=m}^{\infty} p_l\, z^l \sum_{k=i+1}^{\infty} h_{k-l} z^{k-l}\end{aligned}$$

$$= \sum_{l=0}^{m-1} p_l\, z^l [H(z) - H_{m-l-1}(z)] + [P(z) - P_{m-1}(z)]\, H(z)$$

$$= P(z)H(z) - \sum_{l=0}^{m-1} p_l\, z^l\, H_{m-l-1}(z) \qquad \text{(C.47)}$$

$$\sum_{k=m}^{\infty} \sum_{i=1}^{m-1} i\, p_i\, g_{k-i+1}\, z^k = \sum_{i=1}^{m-1} i\, p_i\, z^{i-1} \sum_{k=m}^{\infty} g_{k-i+1}\, z^{k-i+1}$$

$$= \sum_{i=1}^{m-1} i\, p_i\, z^{i-1} [\sum_{j=0}^{\infty} g_j\, z^j - \sum_{j=0}^{m-i} g_j\, z^j]$$

$$= G(z)\, P'_{m-1}(z) - \sum_{i=1}^{m-1} i\, p_i\, z^{i-1}\, G_{m-i}(z) \qquad \text{(C.48)}$$

$$\sum_{k=m}^{\infty} \sum_{j=m}^{k} p_j\, g_{k-j+1}\, z^k = \sum_{j=m}^{\infty} p_j\, z^{j-1} \sum_{k=j}^{\infty} g_{k-j+1}\, z^{k-j+1}$$

$$= \frac{1}{z} [\sum_{j=0}^{\infty} p_j\, z^j - \sum_{j=0}^{m-1} p_j\, z^j] [\sum_{l=0}^{\infty} g_l\, z^l - g_0]$$

$$= \frac{1}{z} [P(z) - P_{m-1}(z)]\, [G(z) - g_0] \qquad \text{(C.49)}$$

Substituting equations C.45 through C.49 into equation C.44, isolating $P(z)$, and writing it in terms of $N(z)$ and $D(z)$ from section C.1.1, we obtain the following expressions for $N(z)$, $D(z)$ and K:

$$N(z) = -m\,\mu_\beta\, g_0\, P_m(z) - \lambda_\beta\, z \sum_{i=0}^{m-1} p_i\, z^i\, H_{m-i-1}(z) +$$
$$\mu_\beta\, z\, G(z)\, P'_{m-1}(z) - \mu_\beta\, z \sum_{i=1}^{m-1} i\, p_i\, z^{i-1}\, G_{m-i}(z) -$$
$$m\,\mu_\beta\, P_{m-1}(z)\, [G(z) - g_0] + z(\lambda_\beta + m\,\mu_\beta)\, P_{m-1}(z), \qquad \text{(C.50)}$$

$$D(z) = \lambda_\beta\, z\, [1 - H(z)] + m\,\mu_\beta\, [z - G(z)], \qquad \text{(C.51)}$$

$$K = 1. \qquad \text{(C.52)}$$

Now we turn to the boundary probabilities equation 8.18:

$$(\lambda_\beta + k\,\mu_\beta)\,p_k = (k+1)\,\mu_\beta\,g_0\,p_{k+1} + \lambda_\beta \sum_{l=0}^{k-1} p_l\,h_{k-l} + \mu_\beta \sum_{i=1}^{k} i\,p_i\,g_{k-i+1} \tag{C.53}$$

We are interested in an expression that can be used to simplify equation C.50. We start by multiplying both sides of equation C.53 by z^k, and then we sum from 0 to $m-1$.

$$\begin{aligned}\lambda_\beta \sum_{k=0}^{m-1} p_k\,z^k + \mu_\beta \sum_{k=0}^{m-1} k\,p_k\,z^k &= \mu_\beta\,g_0 \sum_{k=0}^{m-1}(k+1)\,p_{k+1}\,z^k + \\ &\lambda_\beta \sum_{k=0}^{m-1}\sum_{l=0}^{k-1} p_l\,h_{k-l}\,z^k + \\ &\mu_\beta \sum_{k=0}^{m-1}\sum_{i=1}^{k} i\,p_i\,g_{k-i+1}\,z^k \end{aligned} \tag{C.54}$$

Again we examine separately each one of the sums in equation C.54 and write them in terms of the expressions defined in C.40, C.41 and C.42.

$$\sum_{k=0}^{m-1} k\,p_k\,z^k = z\sum_{k=1}^{m-1} k\,p_k\,z^{k-1} = z\,P'_{m-1}(z) \tag{C.55}$$

$$\sum_{k=0}^{m-1}(k+1)\,p_{k+1}\,z^k = \sum_{j=1}^{m} j\,p_j\,z^{j-1} = P'_m(z) \tag{C.56}$$

$$\begin{aligned}\sum_{k=0}^{m-1}\sum_{l=0}^{k-1} p_l\,h_{k-l}\,z^k &= \sum_{l=0}^{m-2} p_l\,z^l \sum_{k=l+1}^{m-1} h_{k-l}\,z^{k-l} \\ &= \sum_{l=0}^{m-2} p_l\,z^l\,H_{m-l-1}(z)\end{aligned} \tag{C.57}$$

$$\sum_{k=0}^{m-1}\sum_{i=1}^{k} i\,p_i\,g_{k-i+1}\,z^k = \sum_{i=1}^{m-1} i\,p_i\,z^{i-1} \sum_{k=i}^{m-1} g_{k-i+1}\,z^{k-i+1}$$

$$
\begin{aligned}
&= \sum_{i=1}^{m-1} i\, p_i\, z^{i-1} \sum_{j=1}^{m-i} g_j\, z^j \\
&= \sum_{i=1}^{m-1} i\, p_i\, z^{i-1}\, [G_{m-i}(z) - g_0] \\
&= -g_0\, P'_{m-1}(z) + \sum_{i=1}^{m-1} i\, p_i\, z^{i-1} G_{m-i}(z) \qquad \text{(C.58)}
\end{aligned}
$$

Substituting equations C.55 through C.58 into equation C.54, and isolating the factors that involve sums, results

$$
\begin{aligned}
&-\lambda_\beta \sum_{l=0}^{m-2} p_l\, z^l\, H_{m-l-1}(z) \;-\; \mu_\beta \sum_{i=1}^{m-1} i\, p_i\, z^{i-1} G_{m-i}(z) \;= \\
&\mu_\beta\, g_0\, [P'_m(z) - P'_{m-1}(z)] \;-\; \lambda_\beta\, P_{m-1}(z) \;-\; \mu_\beta\, z\, P'_{m-1}(z). \qquad \text{(C.59)}
\end{aligned}
$$

Now we can substitute equation C.59 into equation C.50.

$$
\begin{aligned}
N(z) \;=\; & -m\,\mu_\beta\, g_0\, P_m(z) \;+\; \mu_\beta\, g_0\, [P'_m(z) - P'_{m-1}(z)] \;-\; \lambda_\beta\, P_{m-1}(z) \;- \\
& \mu_\beta\, z\, P'_{m-1}(z). - \; m\,\mu_\beta\, g_0\, P_m(z) \;+\; \mu_\beta\, z\, G(z)\, P'_{m-1}(z) \;- \\
& m\,\mu_\beta\, P_{m-1}(z)\, [G(z) - g_0] \;+\; z(\lambda_\beta \;+\; m\,\mu_\beta)\, P_{m-1}(z), \qquad \text{(C.60)}
\end{aligned}
$$

After identical terms with opposite signs are canceled, and the remaining terms are grouped properly, equation C.60 can be expressed as

$$
\begin{aligned}
N(z) \;=\; & \mu_\beta\, g_0\, [z\, P'_m(z) - m\, P_m(z)] \;+ \\
& \mu_\beta\, z\, [m\, P_{m-1}(z) - z\, P'_{m-1}(z)] \;+ \\
& \mu_\beta\, [z\, P'_{m-1}(z) - m\, P_{m-1}(z)]\, [G(z) - g_0]. \qquad \text{(C.61)}
\end{aligned}
$$

From the definition of $P_m(z)$ and $P'_m(z)$ follows that

$$
\begin{aligned}
z\, P'_m(z) - m\, P_m(z) \;&=\; z\, P'_{m-1}(z) + z\, m\, p_m\, z^{m-1} \;- \\
&\qquad (m\, P_{m-1}(z) + m\, p_m\, z^m) \\
&=\; z\, P'_{m-1}(z) - m\, P_{m-1}(z). \qquad \text{(C.62)}
\end{aligned}
$$

Substituting this result in equation C.61 and canceling similar terms, results

$$N(z) = \mu_\beta\,[z - G(z)]\,[m\,P_{m-1}(z) - z\,P'_{m-1}(z)]. \tag{C.63}$$

Replacing , $P_{m-1}(z)$ and $P'_{m-1}(z)$ by their definitions given in equation C.40, we obtain

$$N(z) = \mu_\beta\,[z - G(z)]\,\sum_{k=0}^{m-1}(m-k)\,p_k\,z^k. \tag{C.64}$$

Substituting equations C.51, C.52, and C.64 in expression C.24, we obtain the generating function for the number of tokens in the system with m β-units.

$$P(z) = \frac{\mu_\beta\,[z - G(z)]\,\sum_{k=0}^{m-1}(m-k)\,p_k\,z^k}{\lambda_\beta\,z\,[1 - H(z)]\;+\;m\,\mu_\beta\,[z - G(z)]} \tag{C.65}$$

Note that with $m = 1$, expression C.65 is identical to equation C.11, the generating function for the single β-unit system.

The extra linear equation that we need to obtain the value of the boundary probabilities is derived from the observation that the sum of all probabilities p_k must be equal to one. Therefore the limit of $P(z)$ as z goes to 1 must be equal to 1. The direct extraction of this limit in equation C.65 results in an indefinite result, requiring application of L'Hospital's rule.

$$\lim_{z\to 1} P(z) \;=\; \frac{\mu_\beta\,[1-\overline{G}]\,\sum_{k=0}^{m-1}(m-k)p_k}{m\,\mu_\beta\,(1-\overline{G}) - \lambda_\beta\,\overline{H}} \;=\; 1 \tag{C.66}$$

Isolating the sum of p_k, results

$$\sum_{k=0}^{m-1}(m-k)p_k \;=\; \frac{m\,\mu_\beta\,(1-\overline{G}) - \lambda_\beta\,\overline{H}}{\mu_\beta\,(1-\overline{G})}. \tag{C.67}$$

Using an adequate definition for ρ_m, we can write

$$\sum_{k=0}^{m-1}(m-k)\,p_k = m\,(1-\rho_m), \tag{C.68}$$

$$\rho_m = \frac{\lambda_\beta\,\overline{H}}{m\,\mu_\beta\,(1-\overline{G})}. \tag{C.69}$$

Equation C.69 is identical to equation 8.21 of Chapter 8.

C.2.2 Average Number of Tokens in the Multiple β-Unit System

To obtain the average number of tokens in the system, we must again compute the limit of the first derivative of $P(z)$ as z goes to 1. From equations C.51 and C.64 we observe that both the limit of the numerator and denominator of $P(z)$ are equal to 0, therefore with the condition of equation C.29 satisfied, we can use expression C.33 to compute the limit. The necessary derivatives and their limits are provided below:

$$D'(z) = \lambda_\beta\,[1-H(z)-z\,H'(z)] + m\,\mu_\beta\,[1-G'(z)] \tag{C.70}$$

$$\lim_{z\to 1} D'(z) = m\,\mu_\beta\,(1-\overline{G})\,(1-\rho_m) \tag{C.71}$$

$$D''(z) = -\lambda_\beta[2\,H'(z)+zH''(z)] - m\,\mu_\beta\,G''(z) \tag{C.72}$$

$$\lim_{z\to 1} D''(z) = -\lambda_\beta\,[\overline{H^2}+\overline{H}] - m\,\mu_\beta\,(\overline{G^2}-\overline{G}) \tag{C.73}$$

$$N'(z) = \mu_\beta\,[1-G'(z)]\sum_{k=0}^{m-1}(m-k)\,p_k\,z^k + \mu_\beta\,[z-G(z)]\sum_{k=0}^{m-1}k\,(m-k)\,p_k\,z^{k-1} \tag{C.74}$$

$$\lim_{z\to 1} N'(z) = \mu_\beta(1-\overline{G})\sum_{k=0}^{m-1}(m-k)\,p_k \tag{C.75}$$

Replacing equation C.67 into equation C.75, yields

$$\lim_{z\to 1} N'(z) \;=\; m\,\mu_\beta\,(1-\overline{G})\,(1-\rho_m) \tag{C.76}$$

$$\begin{aligned} N''(z) \;=\;& -\mu_\beta\,G''(z)\sum_{k=0}^{m-1}(m-k)\,p_k\,z^k \;+ \\ & 2\,\mu_\beta\,[1-G'(z)]\sum_{k=0}^{m-1}k\,(m-k)\,p_k\,z^{k-1} \;+ \\ & \mu_\beta\,[z-G(z)]\sum_{k=0}^{m-1}(k-1)\,k\,(m-k)\,p_k\,z^{k-2} \end{aligned} \tag{C.77}$$

$$\begin{aligned} \lim_{z\to 1} N''(z) \;=\;& -\mu_\beta(\overline{G^2}-\overline{G})\sum_{k=0}^{m-1}(m-k)\,p_k \;+ \\ & 2\,\mu_\beta(1-\overline{G})\sum_{k=0}^{m-1}k\,(m-k)\,p_k \end{aligned} \tag{C.78}$$

Replacing equation C.67 into equation C.78, we obtain

$$\begin{aligned} \lim_{z\to 1} N''(z) \;=\;& -m\,\mu_\beta(\overline{G^2}-\overline{G})\,(1-\rho_m) \\ & 2\,\mu_\beta(1-\overline{G})\sum_{k=0}^{m-1}k\,(m-k)\,p_k \end{aligned} \tag{C.79}$$

Observing that

$$\lim_{z\to 1} D'(z) \;=\; \lim_{z\to 1} N'(z) \tag{C.80}$$

We can rewrite equation C.33 as

$$\overline{N}(m) = \lim_{z\to 1} P'(z) \;=\; \frac{K}{2}\,\lim_{z\to 1}\frac{N''(z)}{D'(z)} \;-\; \frac{D''(z)}{D'(z)} \tag{C.81}$$

Remembering that we chose the value of $K = 1$ when we defined $N(z)$ and $D(z)$ for the multiple β-unit system, we can replace equations C.71, C.73, C.76 and C.79 into equation C.81 and canceling identical terms results in:

$$\overline{N}(m) = -\frac{(\overline{G^2}-\overline{G})}{2(1-\overline{G})} + \frac{\sum_{k=0}^{m-1} k(m-k)p_k}{m(1-\rho_m)} + \frac{\lambda_\beta(\overline{H^2}+\overline{H})}{2m\mu_\beta(1-\overline{G})(1-\rho_m)} + \frac{(\overline{G^2}-\overline{G})}{2(1-\overline{G})(1-\rho_m)} \quad \text{(C.82)}$$

Grouping identical terms in equation C.82, we obtain the final expression for $\overline{N}(m)$:

$$\overline{N}(m) = \frac{\rho_m}{2(1-\rho_m)}\left[\frac{(\overline{H^2}+\overline{H})}{\overline{H}} + \frac{(\overline{G^2}-\overline{G})}{(1-\overline{G})}\right] + \frac{\sum_{k=0}^{m-1} k(m-k)p_k}{m(1-\rho_m)}. \quad \text{(C.83)}$$

This is the expression for the average number of tokens in a system with m β-units (eq. 8.24 of chapter 8). When $m = 1$ equation C.83 becomes identical to equation C.39 as expected.

Bibliography

[1] A. Acharya, *Design of PPL: A parallel production language*. School of Computer Science, Carnegie Mellon University, Preliminary draft., 1993.

[2] A. Acharya and M. Tambe, *Collection-oriented match: Scaling up the data in production systems*. Tech. Rep. CMU-CS-92-218, Carnegie-Mellon University, Pittsburgh, December 1992.

[3] A. Acharya, M. Tambe, and A. Gupta, *Implementation of production systems on message-passing computers*, in IEEE Trans. on Parallel and Distributed Systems, vol. 3, July 1992, pp. 477–487.

[4] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley Pub. Co., Massachusetts, 1986.

[5] A. O. Allen, *Probability, Statistics, and Queueing Theory with Computer Science Applications*, Academic Press, Boston, 1990.

[6] J. N. Amaral and J. Ghosh, *Speeding up production systems: From concurrent matching to parallel rule firing*, in Parallel Processing for AI, L. N. Kanal, V. Kumar, H. Kitani, and C. Suttner, eds., Elsevier Science Publishers B.V., 1994, ch. 7, pp. 139–160.

[7] F. Barachini and N. Theuretzbacher, *The challenge of real-time process control for production systems*, in Proceedings of National Conference on Artificial Intelligence, August 1988, pp. 705–709.

[8] R. Bechtel and M. C. Rowe, *Parallel inference performance prediction*, in Proceedings of the IEEE 1990 National Aerospace and Electronics

Conference - NAECON, May 1990, pp. 21–25.

[9] R. A. Brooks, *Intelligence without reason*, in Proceedings of the International Joint Conference on Artificial Intelligence, August 1991, pp. 569–595.

[10] B. G. Buchanan and E. H. Shortliffe, *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley Pu. Co., Reading, MA, 1984.

[11] J.-F. Chang and Y.-P. Harn, *A discrete-time priority queue with two-class customers and bulk services*, Queueing Systems, 10 (1992), pp. 185–212.

[12] A. Clark, *Microcognition: Philosophy, Cognitive Science, and Parallel Distributed Processing*, A Bradford Book, Cambridge,MA, 1991.

[13] T. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, The MIT Press, Cambridge, 1991.

[14] G. Cowley, *The rise of cyberdoc*, Newsweek, (1994), pp. 54–55.

[15] H. G. Cragon, *Memory systems and pipelined processors.* Preliminary draft, February 1994.

[16] J. M. Crawford and B. Kuipers, *Towards a theory of access-limited logic for knowledge representation*, in Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning, 1989.

[17] P. S. de Souza, *Asynchronous Organizations for Multi-Algorithm Problems*, PhD thesis, Carnegie Mellon University, Pittsburgh, April 1993.

[18] M. K. El-Najdawi and A. C. Stylianou, *Expert support systems: Integrating AI technologies*, Communications of the ACM, 36 (1993), pp. 55–65.

[19] R. E. Fikes, *REF-ARF: A system for solving problems stated as procedures*, Artificial Intelligence, 1 (1970), pp. 27–120.

[20] C. L. Forgy, *On the Efficient Implementations of Production Systems*, PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1979.

[21] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, 1979.

[22] M. R. Garey, D. S. Johnson, and L. Stockmeyer, *Some simplified NP-complete graph problems*, Theor. Comput. Sci., 1 (1976), pp. 237–267.

[23] J.-L. Gaudiot and A. Sohn, *Data-driven parallel production systems*, IEEE Transactions on Software Engineering, 16 (1990), pp. 281–291.

[24] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, Mass., 1989.

[25] ——, *Genetic and evolutionary algorithms come of age*, Communications of ACM, 37 (1994), pp. 113–119.

[26] D. N. Gordin and A. J. Pasik, *Set-oriented constructs: From rete rule bases to database systems*, in Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, May 1991, pp. 60–67.

[27] A. Gupta, *Implementing OPS5 production systems on DADO*, in Proceedings of International Conference on Parallel Processing, 1984, pp. 83–91.

[28] ——, *Parallelism in Production Systems*, PhD thesis, Carnegie Mellon University, Pittsburgh, PA, March 1986.

[29] A. Gupta, C. Forgy, and A. Newell, *High-speed implementations of rule-based systems*, ACM Transactions on Computer Systems, 7 (1989), pp. 119–146.

[30] A. Gupta, M. Tambe, D. Kalp, C. L. Forgy, and A. Newell, *Parallel implementation of OPS5 on the Encore multiprocessor: Results and analysis*, International Journal of Parallel Programing, 17 (1988).

[31] F. Hayes-Roth and N. Jacobstein, *The state of knowledge-based systems*, Communications of the ACM, 37 (1994), pp. 26–39.

[32] F. Hayes-Roth, D. A. Waterman, and D. B. Lenat, *Building Expert Systems*, Addison-Wesley Pub. Co., Massachusetts, 1983.

[33] S. Haykin, *Neural Networks: A Comprehensive Foundation*, Macmillan College Pub. Co., New York, 1994.

[34] A. Hodges, *Alan Turing: The Enigma*, Simon and Schuster, New York, NY, 1983.

[35] D. R. Hofstadter, *Gödel, Escher, Bach: An Eternal Golden Braid*, Vintage Books, New York, 1989.

[36] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*, MIT Press, Cambridge, Mass., 1992.

[37] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability and Programmability*, McGraw-Hill, New York, 1993.

[38] T. Ishida and S. Stolfo, *Towards the parallel execution of rules in production system programs*, in Proceedings of International Conference on Parallel Processing, 1985, pp. 568–575.

[39] T. Ishida, M. Yokoo, and L. Gasser, *An organizational approach to adaptive production systems*, in Proceedings of National Conference on Artificial Intelligence, July 1990, pp. 52–58.

[40] P. C. Jackson, *Introduction to Artificial Intelligence*, Dover Pub., New York, 1985.

[41] K. Kant, *Introduction to Computer System Performance Evaluation*, McGraw-Hill, New York, 1993.

[42] M. A. Kelly and R. E. Seviora, *An evaluation of DRete on CUPID for OPS5 matching*, in Proceedings of the International Joint Conference on Artificial Intelligence, August 1989, pp. 84–90.

[43] ——, *A multiprocessor architecture for production system matching*, in Proceedings of National Conference on Artificial Intelligence, July 1989, pp. 36–41.

[44] H. Kikuchi, T. Yukawa, K. Matsuzawa, and T. Ishikawa, *Presto: A bus-connected multiprocessor for a Rete-based production system*, in Joint International Conference on Vector and Parallel Processing - CONPAR 90, February 1990, pp. 63–74.

[45] L. Kleinrock, *Queueing Systems Volume I: Theory*, John Wiley & Sons, New York, 1975.

[46] D. E. Knuth, *Fundamental Algorithms*, Addison-Wesley, 1973.

[47] A. J. Kornecki, *AI for air traffic*, IEEE Potentials, 13 (1994), pp. 11–14.

[48] C.-M. Kuo, *Parallel Execution of Production Systems*, PhD thesis, The University of Texas at Austin, Austin, Texas, 1991. Department of Computer Science.

[49] C.-M. Kuo, D. P. Miranker, and J. C. Browne, *On the performance of the CREL system*, Journal of Parallel and Distributed Computing, 13 (1991), pp. 424–441.

[50] S. Kuo and D. Moldovan, *Implementation of multiple rule firing production systems on hypercube*, Journal of Parallel and Distributed Computing, 13 (1991), pp. 383–394.

[51] ——, *Performance comparison of models for multiple rule firing*, in Proceedings of the International Joint Conference on Artificial Intelligence, August 1991, pp. 42–47.

[52] ——, *The state of the art in parallel production systems*, Journal of Parallel and Distributed Computing, 15 (1992), pp. 1–26.

[53] H. S. Lee and M. I. Schor, *Match algorithms for generalized Rete Networks*, Artificial Intelligence, 54 (1992), pp. 249–274.

[54] J. D. C. Little, *A proof of the queueing formula* $l = \lambda w$, Operations Research, 9 (1961), pp. 383–387.

[55] B. J. Lofaso, *Join optimization in a compiled OPS5 environment*, Master's thesis, The University of Texas at Austin, Austin, Texas, December 1988. Department of Computer Science.

[56] R. K. Miller, *The 1984 Inventory of Expert Systems*, SEAI Institute, Madison, GE, 1984.

[57] M. Minsky, *The Society of Mind*, Simon & Schuster Inc., New York, 1986.

[58] D. P. Miranker, *TREAT: A New and Efficient Match Algorithm for AI Production Systems*, Pittman/Morgan-Kaufman, 1990.

[59] D. I. Moldovan, *Rubic: A multiprocessor for rule-based systems*, IEEE Transactions on Systems, Man and Cybernetics, 19 (1989), pp. 699–706.

[60] S. Murthy, *Synergy in Cooperating Agents: Designing Manipulators from Task Specifications*, PhD thesis, Carnegie-Mellon University, September 1992.

[61] P. Nayak, A. Gupta, and P. Rosenbloom, *Comparison of the Rete and Treat production matchers for SOAR (a summary)*, in Proceedings of National Conference on Artificial Intelligence, August 1988, pp. 693–698.

[62] D. E. Neiman, *Control issues in parallel rule-firing production systems*, in Proceedings of National Conference on Artificial Intelligence, July 1991, pp. 310–316.

[63] ——, *Design and Control of Parallel Rule-Firing Production Systems*, PhD thesis, University of Massachusets, Amherst, MA, September 1992.

[64] K. Oflazer, *Partitioning in parallel processing of production systems*, in Proceedings of International Conference on Parallel Processing, 1984, pp. 92–100.

[65] J. Palfreman, *The machine that changed the world.* TV Series produced by WGBH Boston/BBC TV, Broadcast by PBS, 1992.

[66] Z. Pylyshyn, *Computation and Cognition*, MIT Press, Cambridge, MA, 1986.

[67] E. Rich and K. Knight, *Artificial Intelligence*, McGraw-Hill, Reading, Mass., 1991.

[68] D. Riecken, *Intelligent agents*, Communications of the ACM, 37 (1994), pp. 18–21.

[69] M. C. Rowe, J. Labhart, R. Bechtel, S. Matney, and S. Carrow, *Forward chaining parallel inference*, in Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing, December 1990, pp. 455–462.

[70] J. Schmolze, *A parallel asynchronous distributed production system*, in Proceedings of National Conference on Artificial Intelligence, 1990, pp. 65–71.

[71] J. G. Schmolze, *Guaranteeing serializable results in synchronous parallel production systems*, Journal of Parallel and Distributed Computing, 13 (1991), pp. 348–365.

[72] J. G. Schmolze and W. Snyder, *Using confluence to control parallel production systems*, in Second International Workshop on Parallel Processing for Artificial Intelligence (PPAI-93), August 1993.

[73] F. Schreiner and G. Zimmermann, *PESA I - a parallel architecture for production systems*, in Proc. 1986 International Conference of Parallel Processing, 1986.

[74] O. G. Selfridge, *Pandemonium: A paradigm for learning*, in Proceeding of a Symposium Held at the National Physical Laboratory, November 1958, pp. 513–526. Reprinted in *Neurocomputing - Foundations of Research*, edited by J. A. Anderson and R. Rosenfeld, The MIT Press, 1988.

[75] J. E. Smith and A. R. Pleszkum, *Implementation of precise interrupts in pipelined processors*, in Proc. 12th Annual International Symposium on Computer Architecture, June 1985, pp. 36–44.

[76] S. W. Smoliar, *Book review: Marvin Minsky, the society of mind*, Artificial Intelligence, 48 (1991), pp. 349–370.

[77] G. S. Sohi and S. Vajapeyam, *Instruction issue logic for high-performance, interruptible pipeline processors*, in Proc. 14th Annual International Symposium on Computer Architecture, 27-34, p. 1987.

[78] A. Sohn and J.-L. Gaudiot, *A macro actor/token implementation of production systems on a data-flow multiprocessor*, in Proceedings of the International Joint Conference on Artificial Intelligence, August 1991, pp. 36–41.

[79] J. Srivastava, J.-H. Wang, and W. T. Tsai, *A transaction model for parallel production systems: Part i. model and algorithms*, International Journal on Artificial Intelligence Tools, 2 (1993), pp. 395–429.

[80] S. J. Stolfo, D. Miranker, and D. E. Shaw, *Architecture and applications of DADO: A large-scale parallel computer for artificial intelligence*, in Proceedings of the International Joint Conference on Artificial Intelligence, August 1983, pp. 850–854.

[81] S. J. Stolfo and D. P. Miranker, *DADO: A parallel processor for expert systems*, in Proceedings of International Conference on Parallel Processing, April 1984, pp. 74–82.

[82] A. A. Stolfo *et al*, *A high bandwidth intelligent memory for supercomputers*, in Proceedings of Supercomputing Conference, May 1988, pp. 517–524.

[83] S. J. Stolfo *et al*, *PARULEL: Parallel rule processing using meta-rules for redaction*, Journal of Parallel and Distributed Computing, 13 (1991), pp. 366–382.

[84] A. C. Stylianou, G. R. Madey, and R. D. Smith, *Selection criteria for expert system shells: A socio-technical framework*, Communications of the ACM, 35 (1992), pp. 30–48.

[85] S. N. Talukdar and P. S. de Souza, *Scale efficient organizations*, in Proc. of IEEE International Conference on Systems, Man and Cybernetics, October 1992, pp. 1458–1463.

[86] C. P. Thacker, D. G. Conroy, and L. C. Stewart, *The alpha demonstration unit: A high-performance multiprocessor*, Communications of the ACM, 36 (1993), pp. 55–67.

[87] A. Turing, *Computing machine and intelligence*, Mind, (1950).

[88] J. P. Wade, *An integrated content addressable memory system*, PhD thesis, Massachusetts Institute of Technology, May 1988.

[89] J.-H. Wang, J. Srivastava, and W. T. Tsai, *A transaction model for parallel production systems: Part ii. model and evaluation*, International Journal on Artificial Intelligence Tools, 2 (1993), pp. 431–457.

[90] J. A. White, J. W. Schmidt, and G. K. Bennett, *Analysis of Queueing Systems*, Academic Press, New York, 1975.

[91] S. Wu and J. C. Browne, *Explicit parallel structuring for rule based programming.* Dept. of Computer Science - The University of Texas at Austin, June 1993.

[92] J. Xu and K. Hwang, *Mapping rule-based systems onto multicomputers using simulated annealing*, Journal of Parallel and Distributed Computing, 13 (1991), pp. 442–455.

[93] T. Yukawa, T. Ishikawa, H. Kikuchi, and K. Matsuzawa, *TWIN: A parallel scheme for a production system featuring both control and data parallelism*, in Proceedings of the 7th Conference on Artificial Intelligence Applications, February 1991, pp. 64–70.

[94] L. A. Zadeh, *Fuzzy logic, neural networks, and soft computing*, Communications of ACM, 37 (1994), pp. 77–84.

Vita

José Nelson Amaral was born in São Luiz Gonzaga, Rio Grande do Sul (RS), Brazil, on January 30, 1964, the son of Nelson Moraes Amaral and Dioraci Terezinha Rambo Amaral. He graduated in Electrical Engineering from Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), Porto Alegre, RS, Brazil, in August, 1987, finishing second in his class. Following graduation he accepted a position at PUCRS, teaching undergraduate laboratory classes as an auxiliary professor. During this period he also worked as consultant for a Brazilian computer manufacturer, EDISA - Eletronica Digital SA. In the following year, he was awarded a competitive government fellowship from CAPES[1] with paid leave from PUCRS to pursue a Master's Degree at Instituto Tecnológico de Aeronáutica (ITA), São José dos Campos, São Paulo, Brazil. He received his Master of Engineering Degree from ITA in October 1989, graduating with honors, and received a commendation for his thesis work from the examining committee.

In November 1989 he resumed teaching graduate and undergraduate classes at PUCRS and served as teacher and coordinator for extension courses to the local industrial community. In 1990 he was awarded a competitive fellowship from CNPq[2] with paid leave from PUCRS to pursue his PhD degree and joined the graduate program of The University of Texas at Austin.

[1] *Coordenadoria de Aperfeiçoamento de Pessoal do Ensino Superior*, a Brazilian government agency that finances the improvement of high level education. This fellowship was part of the *Programa para Incentivo à Capacitação de Docentes* (PICD), a government program for the qualification of university professors.

[2] *Conselho Nacional de Desenvolvimento Científico e Tecnológico*, a Brazilian government agency for the financing of scientific and technological research.

Permanent address: Rua Jacinto Gomes, 326 ap. 12
90.040-Porto Alegre-RS
Brazil

This dissertation was typeset by the author using the LaTeX document preparation system.