Order Number 9304677

An operational approach to computer system specification using applicative high-order logic

Jin, Ti, Ph.D.

City University of New York, 1992

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

AN OPERATIONAL APPROACH TO COMPUTER SYSTEM SPECIFICATION USING APPLICATIVE HIGH ORDER LOGIC

by

TI JIN

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

1992

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

Sept 30, 1992
Date

Chair of Examining Committee

Sept 30, 1992
Date

Executive Officer

Professor Seyed-Ali Ghozati
Professor Syed Ahamed
Professor Michael Houlihan

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

AN OPERATIONAL APPROACH TO COMPUTER SYSTEM SPECIFICATION USING APPLICATIVE HIGH ORDER LOGIC

by

TI JIN

Advisor: Professor Stanley Habib

The objective of this thesis is to present and demonstrate the concept of applying an applicative high order logic to a fully-operational specification for complex computer system design and early implement-free simulation. Two realms are literally involved in this research. First, a Process-oriented, Applicative, Interpretable Specification Language, short for "PAISLey," is proposed. An applicative specification serves as an operating paradigm to embody an automatic test and check of design correctness, system consistency, to carry out faithfully an instance-free design simulation, gather significant performance measurements, and so forth. A variety of examples spanning many abstract levels of computer architecture have been exhibited to illuminate how well these subjects can be undertaken.

Secondly, we introduce a newly developed hierarchical cache organization for a tightly-coupled multi-processor system using a single shared memory. A small and fast set-associative memory, which functions as a write multi-buffer, is placed between the main memory and the private caches of local processors in order to resolve the cache coherence, shorten the write through latency, and then reduce memory bus traffic. We name it a Write Look-Ahead Buffer, or a

WLAB. Two generic configurations of this Write Look-Ahead Buffer regarding local caches are discussed. The structure and size of the Write Look-Ahead Buffer will significantly affect the system performance. Several possible cache coherence protocols based on the WLAB organization are also addressed because the selection of the coherence strategy exerts instant effects on its performance.

A considerably large portion of this dissertation is devoted to producing the PAISLey specification for the WLAB cache system. Trace simulation of the WLAB cache system using the PAISLey specification is carried over on SUN SPARC II workstations. A large volume of analytic data is collected directly from running the WLAB specification. Through this exploration of PAISLey to the WLAB cache system, we manifest and verify our original concept of consigning applicative high order logic specifications to the operational computer system description and instance-free simulation.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles - cache memories; C.4 [**Computer Systems Organization**]: Performance of Systems - design studies; measurement techniques; modeling techniques; D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications - languages; methodologies; I.6.5 [**Simulation and Modeling**]: Model Development - modeling methodologies

General Terms: Design, Experimentation, Performance, Languages, Logic Programming

Additional Key Words and Phrases: Architecture, multi-processor, cache coherence, hardware descriptions, applicative (functional) programming, executable specifications, system validation, PAISLey

ACKNOWLEDGEMENT

The accomplishment of this thesis has been an arduous endeavor and has been nourished by many people. First of all, professor Stanley Habib, my mentor, gave me endless patient understanding, constructive advice, enthusiastic encouragement, and prompt help, without which the work could not have been completed. I also would like to thank the members of my examining committee, professor Seyed-Ali Ghozati of Queens College, professor Syed Ahamed of the College of Staten Island and professor Michael Houlihan of Fordham University, for their time in reviewing the draft and serving on the committee.

I am greatly indebted to many faculty members of CCNY for their invaluable help. Professor Daniel McCracken, ex-Chairman, and professor Gary Bloom, Chair of the Department of Computer Science, have supplied me with much-needed research facilities and a pleasant working environment. Professor Michael Anshel showed the strongest confidence in my capability: talking with him has always inspired me to greater efforts. Professor Octavio Betancourt had been supportive and helpful in the earlier phase of my doctoral study. Professor Stephen Lucci is the first faculty member whom I ever met in CCNY and who is also the first person to help me study and teach at CCNY.

I would like to express my gratitude to Nikolaos Papavassiliou. I cannot image how the simulation would have gone on without his assistance in setting up SUN workstations and putting PAISLey in the LAN server. His knowledge of UNIX saved me innumerable times from frustrating system trial and error.

Special thanks go to Dr. Nancy Showers for her support and consideration; particularly, for her

kindness in inviting me to use the LaserJet III Si to produce so beautifully my visuals.

Joseph Driscoll helped me to edit this last version of the thesis. His laborious effort saved me from embarrassment by fixing typos, grammatical bugs and slip-ups in previous drafts. He is also the person, as Doctoral Program Assistant, who provided me with much productive information and suggestions to ease my work.

I have had the great fortune to have many superb college-mates and friends. Especially, I would like to share this honor with John C. Yu, with whom I shared countless days and nights of hard studies and stimulating conversations; Lesen and Yizhi Wang, five-year apartment-mates, who alleviated me of some of life's hardships; Xiaochu Wu, whose heartening help is indelible. We, together, spent, struggled, and surmounted the trials of our first days in America. I am also obliged to Yuhui Li who typed a portion of the appendices.

Certainly, I feel much thankfulness to my family at this special time. My father gave me an early education that would have been, otherwise, impossible during the Great Cultural Revolution. My mother's trust and love perpetually endow me with bravery, determination and the endurance to overcome obstructions.

Finally, I want to dedicate the thesis to my grandfather, Weixian. His spiritual prediction has always been encouraging me to accomplish this commitment. Once upon a time, having read the palm of his five-year-old grandson, he told his son, my father, "one day, this boy will be a doctor ..."

CONTENTS

Abstract iv

Acknowledgement vi

Figures And Tables x

1. Introduction 1

2. Computer System Specifications And Hardware Description Languages 9

2.1 Specification and CHDLs 9

2.2 Executable specification and operating model 15

2.3 System specifications and abstract levels 17

2.4 Applicative (functional) languages 22

2.5 Tolerance of incompleteness 25

2.6 Timing constraints and early performance evaluation 27

2.7 Comparison of PAISLey to other CHDLs 29

3. PAISLey Specification Models 36

3.1 An overview of PAISLey 36

3.2 Specification of multi-level computer systems using PAISLey 42

3.2.1 Specification in register transfer and logic design level 42

3.2.2 Specification in higher architectural level 48

3.2.3 Specification of asynchronous interactions 55

3.3 Performance simulation by execution of the PAISLey specification 62

3.3.1 Time constraints in PAISLey 62

3.3.2 Running methods and scheduling strategies 66

3.3.3 Consistencies and performance measurements 69

4. The WLAB Cache System **75**

4.1 The WLAB cache memory organization 77

4.2 Configurations of the Write Look-Ahead Buffer 82

4.3 The WLAB cache coherence protocol 87

4.4 Cost and performance expectation 91

5. Specification And Performance Of The WLAB Cache Systems **96**

5.1 The PAISLey specification of a WLAB system 96

5.2 Simulation and performance evaluation 103

6. Conclusion And Future Research **123**

Appendix A: A Quick Survey Of CHDLs **126**

Appendix B: Specification Of A WLAB System For 4-Multiprocessors **135**

Appendix C: A Grammar For The PAISLey Language **151**

Appendix D: The PAISLey Interpreter Commands **159**

Bibliography **161**

FIGURES AND TABLES

Figure 2.1	Abstract Levels of Computer Architecture	20
Figure 3.1	Specification 1: A 50MHz Clock Generator	37
Figure 3.2	Trace of the Execution of Specification 1	39
Figure 3.3	Specification 2: Refinement of Specification 1	40
Figure 3.4	Trace of the Execution of Specification 2	41
Figure 3.5	Specification 3: A Modulo-8 Counter	43
Figure 3.6	Specification 4: A General Logic Function f(w, x, y, z)	45
Figure 3.7	Bit-Wise Logical Operations	46
Figure 3.8	C Process of *n-ary-or*	47
Figure 3.9	Specification 5: Instruction Fetching and Execution Stages	50
Figure 3.10	Specification 5 (Continued)	51
Figure 3.11	Specification 6: A Microprogrammed Architecture	52
Figure 3.12	Match Relations of Exchange Types	57
Figure 3.13	Synchronization Patterns of Exchange Functions	58
Figure 3.14	Specification 7: A Synchronizing Mechanism - Semaphore	60
Figure 3.15	A Mutual Activity Model	61
Figure 3.16	Specification 8: A Template Model of System Mutual Activities	61
Figure 3.17	Time Constraints in PAISLey	64
Figure 3.18	Processes in Action	68
Figure 3.19	Trace of the Execution of the Modulo-8 Counter Specification	71
Figure 3.20	Trace of the Execution that Detects Performance Deficiency	72
Figure 3.21	Trace of the Execution of Specification 7 - A Synchronizing Mechanism	73
Figure 4.1	The Organization of a WLAB Cache System and Hardwired Semaphores	78
Figure 4.2	Finite Automata for the Status of Semaphores	82
Figure 4.3	Configurations of the WLAB with Respect to Local Caches	83

Figure 4.4	Set and Tag Mapping in Diagram b of Figure 4.3	85
Figure 4.5	State Transition Diagram for the WLAB Cache Coherence Protocol	88
Figure 4.6	Typical State Transition Diagrams for Snooping Protocols	90
Figure 5.1	A K-Way Set-Associative Mapping Cache Organization (K=2)	97
Figure 5.2	Trace from Execution of the WLAB Cache System Specification	105
Figure 5.3	Percentage for the WLAB to Have More Than One Word	106
Figure 5.4	Hit Ratio of the WLAB System for Uni-Processor Environment (1)	107
Figure 5.5	Hit Ratio of the WLAB System for Uni-Processor Environment (2)	108
Figure 5.6	Hit Ratio of the WLAB System for Uni-Processor Environment (3)	108
Figure 5.7	Hit Ratio of the WLAB System for Uni-Processor Environment (4)	109
Figure 5.8	Hit Ratio of the WLAB System for Uni-Processor Environment (5)	109
Figure 5.9	Hit Ratio of the WLAB System for Dual-Processor Environment (1)	111
Figure 5.10	Hit Ratio of the WLAB System for Dual-Processor Environment (2)	111
Figure 5.11	Hit Ratio of the WLAB System for Dual-Processor Environment (3)	112
Figure 5.12	Hit Ratio of the WLAB System for Dual-Processor Environment (4)	112
Figure 5.13	Hit Ratio of the WLAB System for Dual-Processor Environment (5)	113
Figure 5.14	Hit Ratio of the WLAB System for Four-Processor Environment (1)	113
Figure 5.15	Hit Ratio of the WLAB System for Four-Processor Environment (2)	114
Figure 5.16	Hit Ratio of the WLAB System for Four-Processor Environment (3)	114
Figure 5.17	Hit Ratio of the WLAB System for Four-Processor Environment (4)	115
Figure 5.18	Hit Ratio of the WLAB System for Four-Processor Environment (5)	115
Figure 5.19	Memory Traffic Ratio of the WLAB Cache Systems	116
Figure 5.20	Write Traffic Reduction of the WLAB Cache Systems	117
Figure 5.21	Memory Traffic Ratio and Write Traffic Saving for Dual-Processors (1)	117
Figure 5.22	Memory Traffic Ratio and Write Traffic Saving for Dual-Processors (2)	118
Figure 5.23	Memory Traffic Ratio and Write Traffic Reduction for Uni-Processor	118
Figure 5.24	The Write Look-Ahead Buffer Utilization and Collision (1)	120
Figure 5.25	The Write Look-Ahead Buffer Utilization and Collision (2)	121
Figure 5.26	The Write Look-Ahead Buffer Utilization and Collision (3)	121

1. INTRODUCTION

The originality of this research revolves around two areas. Firstly, an applicative style executable specification language is exploited to formally specify the properties and functioning of a complex system of the interconnection of computer architecture components. An operational model is constructed directly by using this executable specification with no requirements of any conversion or modification. An automatic checking of system consistency will start immediately by running the specification. Simulation of the specified system can be carried out if no inconsistency is detected. Performance measurements of an early implement-free system model are gathered, directly and conveniently, from traces of execution.

Secondly, an architectural scheme for enhanced memory access comprising the Write Look-Ahead Buffer has been proposed. It is a novel hierarchical memory structure that facilitates both uni-processor and multi-processor systems, and resolves the cache coherence problem smoothly and effectively. How this original idea results in improved memory traffic performance and reduced memory access latency, over similar configurations which use simple one-level or even two-level cache methodologies, is demonstrated. The simulation and performance evaluation is the immediate consequence of application and execution of an operational specification. This work has lead to several papers which are currently submitted for publication and which we hope will prove to be a substantial contribution to these areas of computer architecture design, formal specification, simulation and performance evaluation.

It is now widely recognized that a formal specification for a computer system has a tremendous and powerful impact on the quality, high-performance, longevity of the ultimate product, and the efficiency and manageability of its further development. It is because a specification can be used

as a well-defined communicating document, an analytic model, or can even be systematically converted to a simulation model of the computer system being designed. More and more manufacturers realize that complex computer systems should be carefully and unambiguously specified before too much effort has been expended in the design and implementation. A formal specification will significantly reduce the time of system design and save costs.

The first part of this thesis explores how an operational and applicative specification can be applied to the description of computer systems. By "operational," we mean that the specification can be executed directly, and a simulation, then a feasibility test, correctness inspection, consistency checking and performance evaluation may be accomplished immediately from the results of the execution. By "applicative," we mean the specification, using high order logic programming, can be viewed as a black box, and carried out in a top-down style. That is, the specification starts at a very high architectural or behavioral level, and is then refined to a desired detailed level, even to a very detailed register-transfer or logical gate level.

PAISLey [Zav82a, 87a, 87b] is selected to embody and serve as a paradigm for this operational specification. PAISLey, standing for Process-oriented, Applicative (Functional), Interpretable Specification, was developed by P. Zave, together with a group working at AT&T Bell Laboratories. It is implemented under a variety of UNIX™ systems such as System V, BSD™ 4.2 & 4.3, SunOS™, Ultrix™, etc. Expanding upon earlier examples, described in the literature as computing models of real-time systems, we suggest in this paper that PAISLey is well suited for possible use as an executable computer hardware and system description language.

Lead by VHDL [Ayl86, IEE87, LipsM86, NasS86, ShaL85], a hardware description language for very high speed integrated circuits, there are a fairly large number of hardware description

---

™ UNIX is a registered trademark of AT&T Bell Laboratories. BSD is a trademark of University of California, Berkeley. SunOS is a trademark of Sun Microsystems, Inc. Ultrix is a registered trademark of Digital Equipment Corporation.

languages in literature [BarbK87, BorW91, DarR89, Das84a, Hart87a]. Unfortunately, very few of them are used in actual production in manufacturing industries. Amongst various reasons for this situation, two, which we are considering in this text, are essential. The first is that most of the hardware description languages tend to be complete and complex, and lead to a lengthy and cumbersome specification. The other is lack of direct executability. This deficiency requires special supporting tools to be developed for making the original specification first converted into executable form and then simulated. This may cause deviation from the genuine design, no matter whether this conversion is done manually by experts or automatically in the internal system. Both of these deficiencies cause the formal specification to lose its originally concise and intuitive advantages and interfere with the wide acceptance and popular use in productive industry. Computer hardware description languages, or CHDLs, like VHDL are also known for deficiencies in specifying system synthesis and instance-free design [Agn91].

PAISLey, by contrast, without using a complicated mechanism, provides an accurate, unambiguous, complete and executable specification. We have no intention of adding any more languages to the already crowded world of the computer hardware description languages. A vital subject here is to introduce the concept of combining a formal applicative specification and an operational simulation model into one stage. Three advantages will result: (1) distortion and deviation resulting from converting a formal specification to an executable simulation will be completely eliminated; (2) system measurements will be collected directly from execution of the specification and a truly early performance evaluation can be conducted accordingly, and; (3) the length of design cycle will also be shortened remarkably by carrying out an earlier performance simulation on the executable specification. PAISLey has been successfully used as a formal specification approach in a wide spectrum of real-time systems [Bru86, LippM88, McCZ89, Surl86, Zav89, Zav91]. We choose it to instantiate our belief because of its simplicity, conciseness, intuitiveness and executability. By adding a few utilities, it is easily and naturally extended to special uses for the computer system specification and hardware description.

A number of examples are furnished to illustrate the versatile capabilities of PAISLey in specifying multi-level computer architecture. A PAISLey specification of our newly-proposed design of a cache architecture and coherence protocol for the shared memory multiprocessor system has been carefully developed. It has been tested and executed extensively to verify and support our initial design concept. The first portion of this work will focus not only on demonstrating valid formal methods for unambiguous, correct specification, but will also demonstrate certain performance measurements which can be made by coding and executing the specification. In fact, it is the latter that makes our research more significant and useful in practice.

The second part of this research studies a novel cache memory system - the WLAB, an acronym for the Write Look-Ahead Buffer. A Write Look-Ahead Buffer cache system is a hierarchical memory system that we propose for the application of a tightly-coupled multiprocessor system using a single shared memory. A small and fast multi-way set-associative memory, which acts as a write multi-buffer, is inserted between the shared memory and the local caches of processors to decrease the miss latency and write through penalty, and to raise memory bus bandwidth as well. We name it a Write Look-Ahead Buffer, or WLAB for abbreviation. Unlike the regular write multi-buffer that normally complicates the cache coherence problem, the associative searching feature of this new write buffer is able to resolve this dilemma effectively and smoothly. As a reward, the introduction of the Write Look-Ahead Buffer will further reduce the read miss latency by directly reading those blocks from the WLAB instead of the shared memory. A cache coherence protocol based on this organization is presented, too. It differs from others in that a block of the private caches in a local processor has only two regular states, that is, valid or dirty. Without new states expanded just for data coherence, it simplifies the working process of the cache controller for "state testing and decision making". In addition, hardwired local semaphores are introduced to this organization to loosen the contention for the memory bus and take the bus synchronizing burden off system programmers.

Two generic configurations of this Write Look-Ahead Buffer regarding local caches are discussed. Actually, as shown by our simulation results, only the size of the Write Look-Ahead Buffer and the dispatch ratio of the access time of main memory to that of cache will significantly affect hit ratio, the penalty of the miss manipulation, the write block collision, memory bus traffic, write traffic reduction and utilization of the WLAB. The structures of the WLAB itself, as we discuss here, do not contribute a great deal of meaningful scores to these aspects. This research describes in details how this novel memory hierarchical structure is organized, and how simulation and performance evaluation is conducted via execution of an operational specification. Merits of this memory organization will be discussed by presenting analysis results from execution of its PAISLey specification.

We would like to clear up in this introduction one frequently neglected issue in the current literature: that a system specification offers a formal description of a computer system, not merely a hardware description as the term "hardware description languages" often implies. A pure hardware description mainly considers logic and circuit implementation of the designing target, while system specification has to deal with generic architectural organization; interactions between hardware and software, and; interactions between the computer system and its surrounding world. It has to balance overall system performance and make compromises between various system level modules and components, which also depend on the feedback from early system simulation and performance analysis. These requirements make a direct execution of formal specification as a simulation model more favorable because it is more flexible and adjustable during the process of decision making and testing. Unfortunately, these aspects have been largely ignored, not drawing much if any deserved attention in previous work about computer hardware description and specification.

The following paragraphs are going to provide an overview of the dissertation. Section 1 is this introduction itself. Section 2 discusses the computer system specifications and hardware description languages. In particular, Section 2.1 describes general requirements for well-organized

specifications and classifies a finer distinction between the system specifications and hardware descriptions by indicating that a system specification should be fully implementation-independent in order to cover as many implementation techniques as possible. Section 2.2 emphasizes specification for a complete computer system and description of the multi-levels of computer architecture. It points out that the interactions of computer hardware with its working environment, usually an operating system or a system control program, cannot be ignored during the system specification. Those activities have exerted powerful influences on the evolution of both computer architectures and software systems with its developing languages. This bidirectional impact will be continued in future design. Section 2.3 gives our usage of the term "operational" with the purpose of clarifying the word's blurred meaning and delineating it from other significant terminology. Basic requirements and potential advantages of an operational specifying approach are discussed here as well. Section 2.4 through 2.6 elucidate a number of salient features that a modern computer description language should have, especially in functional description (2.4), tolerance of incomplete specification (2.5), and time constraints distributed to real-time components (2.7). Section 2.7 compares PAISLey to other current computer hardware description languages, typically to VHDL - a VHSIC hardware description language initiated and sponsored by DOD. It examines the potential advantages of PAISLey as an operational computer system specifying language.

The whole of section 3 explains how to apply an operational approach to the applicative specification of computer system using the PAISLey language. Section 3.1 presents an overview of the PAISLey language. We make no attempt to offer a complete tutorial text, but simply introduce main features of the PAISLey specifications, exclusively as an operational approach to the computer system description. Accordingly, Section 3.2 enumerates a number of examples for different level computer architecture specifications. Specifically, 3.2.1 provides several specification examples for the register-transfer and logic design level; 3.2.2 uses PAISLey to describe modules in higher architectural levels such as microprogramming and system design levels; 3.2.3 illustrates compact specifications for asynchronous interactivities within computer

systems or between computers and their software developing systems, which we think is more critical to a successful system description. Section 3.3 shows how an executable specification can be employed as a prototype of the objects being designed and be simulated directly to gather instant performance measurements. Here, 3.3.1 tackles the time attributes of the PAISLey specification; 3.3.2 describes running methods and scheduling strategies while demanding execution of the PAISLey specification; 3.3.3 illustrates the application of the PAISLey specification for consistency check and performance evaluation.

Section 4 and 5 represent the second major work of this research and expose the construction and performance evaluation to a newly developed hierarchical memory structure called a WLAB cache system by means of an executable PAISLey specification. Section 4.1 exhibits the organization of the WLAB hierarchical architecture. Section 4.2 deliberates possible configurations of the Write Look-Ahead Buffer in regards to private caches in local processors and presents two generic solutions for its designing and implementing simplicity. Section 4.3 develops a cache coherence protocol based on the organization illustrated in the previous two sections, with an aim to reduce miss latency and processing complexity. Section 4.4 discusses the expectation as referred to cost and performance consideration. Section 5.1 describes the PAISLey specification of the WLAB cache system, which acts as an operative paradigm for simulation. Section 5.2 displays the actual results accumulated from simulation of various WLAB configurations, maneuvered by running the PAISLey specification. The entire performance evaluation is established on the trace simulation, a widely accepted technique in both research and industry to appraise cache effectiveness. A set of well-known SPEC and ATUM traces are used to perform this simulation.

Finally, Section 6 concludes this thesis and talks about plans for continuous studies. Applying an operational applicative specification as a direct simulating paradigm for a collection of system measurements apparently is still in a pioneering stage, and more work needs to be done in the realm. Also, a WLAB cache system is our exciting and fruitful incentive to develop a new computer architecture. We intend, in a long term plan, to expand this simulation extensively, and

attempt to find a way of estimating the optimized parameters which expedite design decision making by means of statistical methods and mathematical tools.

Four appendices are also attached at the end of this thesis. Appendix A supplies us with a quick survey of computer hardware description languages. It stands for our early work when we were preparing this research and determining which language would be chosen for fulfilling the mission of an operational and applicative specification. It is not an exhaustive search: languages are selected on a basis of historical and representative characteristics, ranging from a single-level description languages such as register transfer languages to modern multi-level languages like VHDL, ACE, etc. Appendix B lists a complete PAISLey specification coding for a WLAB cache system based on four processors and a single shared main memory. Utilities written in C and developed to facilitate the processing of large amount of tracing data are also listed there. Finally, Appendix C reprints a reference grammar of the PAISLey language while Appendix D lists the PAISLey interpreter commands by courtesy of Dr. P. Zave*. Three volumes for the complete language and tutorial manuals [Zav87a, b, c] can be acquired from P. Zave upon request.

* Pamela Zave, Room 3D-426, AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

2. COMPUTER SYSTEM SPECIFICATIONS AND HARDWARE DESCRIPTION LANGUAGES

2.1. Specification and CHDLs

A computer system specification is a precise representation of the behavior of a computer system. In its role as a representation for the system design, manufacturers use the specification as a vehicle for communication between customers and designers, or, if within the same corporation, between the product planning division and design division. In other words, a specification is supposed to document a consensus among design staff and patrons. Because of its accuracy and unambiguity, a formal specification can be used as a prototype to testify and evaluate the target system, and eventually decide whether to accept or reject the product in question before too much effort has been expended in its design and implementation. Therefore, it will significantly shorten the design cycle and save expenses. In order to add an auto-testing and operating ability, we strongly recommend an executable specification that allows an immediate check of inner system consistency verification and behavioral simulation, once the object is specified.

A specification must be precise, unambiguous, internally consistent, complete and minimal. The meaning of precision and unambiguity are quite clear. The specification in natural language is sometimes vague and ambiguous. It is for this reason that the formal methods are preferred. An internally inconsistent specification implies that the specified system is not well-defined. In other words, it specifies a system that will be unable to be realized and to function properly. System inconsistency can be easily detected in a formal specification either by a logic reasoning or an auto-checking if it is executable. The word "complete" says that all the requirements and

behaviors are specified, otherwise, an under-constrained specification may not satisfy original requirements of the design problem. "Minimal" signifies that no surplus constraints could be added. The complete and minimal specification guarantees a faithful representation of the target object being designed.

Since a specification is not the design itself, it is rather a truly descriptive behavior of the target system, i.e., it accurately specifies the requirements and functionalities of the systems without under- and over-constraints. A system specification is not exactly the same as a hardware description, though some researchers use these two words interchangeably. The latter is usually, more or less, in reference to implementation techniques and target hardware. This tendency will inevitably yield some behaviors or attach some constraints that do not arise in the original design, because every implementation technique has superfluous constraints and limitations. Meanwhile, the purpose of specifications is to establish an early behavior design and operating model. This is then used to test its internal coherence and the feasibility of its realization. Therefore, a specification must be fully implementation-independent so it can be transformed to many design implementations which are equivalent. We would like to say that a specification is more abstract and oriented to the description of requirements at the system level design. Some authors distinguish by naming them as requirement specification for describing the abstract level, external characteristics and design realization description for concrete level and internal organization [BorW91, Bou91].

In spite of its accuracy requirements with no over and under constraints as mentioned in earlier paragraphs in this text, the following features are considered general requirements for a good specification, based on our view and many others' work [AhtA86, HanD89, Zav84a, ZavS86].

Conciseness - Since it is a communicating vehicle and document, a specification should be as simple as possible. It must be relatively short in length and easy in term of both writing and reading. It should also be understandable and demonstrable. A concise

representation makes a consolidating and integrated requirement specification and prevents people from overshooting the original designing problem occurred by playing the trick and tediously specifying codes and symbols. Unfortunately, most modern powerful and complex CHDLs lack such simplicity.

Intuitiveness - Specifications should be expressed in as direct and native a way as possible, so as to make the specification readable and understandable, while at the same time allowing the intuitive concepts and informal modes of reasoning to be easily related to their formal counter-part. Keeping inherent design senses also avoids the distortion and deviation which may easily happen during the process of converting human thought to a non-well-suitable specifying approach.

Generality - Specification should be written in a fully general way, so that a single specification can be easily and equivalently transformed to many implemention instances. In short, it should be instance-independent and be able to capture easily the desires of designers, but not include implementing details. It is especially important in system level specification as we repeat several time in this paper. Specification oriented to one or two implemention and designing techniques will introduce unnecessary constraints. Besides, abstraction, modularity and reusability are all well-known benefits which profit by generality.

Operationality - Specification must be formally manipulatable if verification is to be expected, or testable if acceptance testing is to be processed, so that it can be determined whether the final product meets the requirements. Ideally, specifications are directly executable, so that engineers need only concentrate on the faithful specification itself, and need not worry about how to perform the equivalent transformation in order to obtain consistency check and performance simulation. The requirements can be truly attached to the executable specification straightforwardly. PAISLey possesses this

excellent quality in spite of the fact that not many CHDLs have such directly operational ability.

Tolerance - It means a tolerance of incompleteness, i.e., a specification should be usable, or executable, in an operational approach, even when it is incomplete, meaning that missing the inner structure of some entities or declaration of other non-relevant processes. That is because specifying a complex computer system in its entirety at one step is unrealistic. Specification is usually carried on via a stepwise refinement approach, some inner nature of entity can be neglected in the first attempt at the abstract level. Furthermore, the inner nature of some entities are really neither known nor are of interest, e.g., we may specify some intervention of a human being as external interruption, but should we specify the human being? Tolerance of incompleteness also adds flexibility to debugging, testing, reusing and retargeting the system being designed. Almost all CHDLs except PAISLey require a full and lengthy definition even for a very high level specification.

Timing - Specifications of computer systems must also be able to characterize the performance constraints. Time constraints usually embody the performance requirements of digital systems as they are real time systems. Therefore, the specification of timing will directly influence the truth of performance analyses in the testing phase. It is more important in the case of an executable specification since the execution itself will be interpreted as a performance simulation.

Parallelism -In multiprocessor and distributed systems, specifications must be capable of expressing highly parallel computations. Otherwise, much of the rich potential of distributed implementations for parallelism can never be realized in most contemporary procedural CHDLs like VHDL because it is extremely difficult to transform a sequential specification into a parallel form. It is constrained by the inherently sequential nature

of the procedural languages. PAISLey provides for specifying maximum parallelism, and asynchronous interaction as it is a non-sequential language.

Most computer system specifications employ computer hardware description languages (CHDLs) as expressive tools. CHDLs, sometimes called hardware description languages (HDLs), are a class of descriptive languages used for the description and design of computer systems. To reduce the describing complexity, a computer architecture is partitioned into several abstract levels. Usually, a CHDL is suitable for one or more level. However, modern CHDLs are required to be well-suited for any abstract level, so that by using the same language, a specification can start at a very high level with very low complexity, and after several refinements, may end up with a more complex and more detailed specification which accurately reflects the target system. Since the first computer hardware description language appeared about thirty-seven years ago, it is commonly called a register transfer language (RTL), hundreds of CHDLs have been published, and dozens of them are continuously added to the list in recent years, even after IEEE announced the adoption of VHDL as a standard of hardware description languages. All of that makes it competitive we are entering this area.

CDL [Chu72a, 72b] probably is one of the oldest hardware description languages at the register transfer level, and has been successfully used in teaching logic design in many academic environments since its inception. ISPS [Barb81, Wilk82] is probably the most influential higher level or behavior description language, whose concepts have had a great impact on development of the later CHDL designs and implementations. STATEMATE [Hare87, HareL88] is a pioneer language that provides a graphic working environment for hardware descriptions. CONLAN [Pil80a, 80b], a consensus language, represents a "family of languages" approach, in which there are "members of CHDLs" in the family with each member language designed only for the specification of one or few abstract levels. Though this has several positive aspects, the methods suffer a number of deficiencies. First, several languages must be understood if two or more abstract levels are involved; second, the member language may be designed so much towards a

specific application, as to lose the generic meaning of the specification. Last, it is difficult to construct the design tools, since several different languages exist in a family. For example, a simulator must accept the specifications in all those member languages. VHDL [IEE87, ShaL85], a high level hardware description language sponsored by the DOD to specify very high speed integrated circuits (VHSIC), represents an opposite approach. In contrast to CONLAN, it uses a single language for multi-level descriptions. VHDL is rich in various descriptive and simulating modules. VHDL, initially sponsored by the DOD as the part of VHSIC program, was approved by IEEE as a standard of CHDLs [IEE87] in 1988. After years of effort, it has been proven that VHDL supports the computer structural design, documentation and efficient simulation of hardware implementation very well; but fails to capture the intuitive intent of designers and scarcely provide a complete technology-free synthesis and implement-independent system specification. However, the complexity and intricacy of descriptive and modeling techniques dwelling in VHDL lead to a lengthy and verbose description, and thus make VHDL specification difficult to learn and understand. That is the reason why many extensions and revisions of VHDL are being developed. In addition, several other specification languages, using completely different methodology, appear to improve expressive capacities specifically for implement-free design and formal verification. Languages using temporal logic and applicative high degree formalism are the most well-known. PAISLey, on the other hand, is an executable and purely applicative specification language which is selected by this work to exploit an operational and functional approach to the specification of complete computer system design. Specification in PAISLey is precise, unambiguous, minimal (without over-constraint), complete (without omitting any requirements), and operational (executable). It is a representative model of the system behaviors and functions. The entire specifying construction and operating methods furnished by PAISLey are referred to as an operational approach. Zave proposed its use for the arena of real-time systems [Zav80, 82a, 84b]. Many conceivable advantages of this operational (executable) specification have been investigated by Zave and Schell [Zav84a, ZavS86]. A comparison of PAISLey with other CHDLs are presented in section 2.7.

Unfortunately, despite the fact that so many CHDLs exist, few computer manufacturers have taken these valuable tools seriously to assist their designs, although many note their importance and significance. The problem is, as indicated by Hartenstein [Hart87b], that the application aspects of existing CHDLs have not been exploited sufficiently, and not enough designers and analysts are sufficiently well trained to use these CHDLs. In addition, most currently-proposed CHDLs will emerge with a large degree of complication and intricacy greatly impeding their acceptance and popular use. This fact inspires us to develop a "state of the art" design and operating model for specifications of computer systems. We make no attempt to design a new descriptive language. This work presents an operational approach using an existing language and we expect to be able to arm the design staff with a functional description.

2.2. Executable Specification and Operating Model

Another very important aspect of system specification is to create a real working and testing environment, then to establish an early simulation model usually requiring an executable specification. An executable specification is a model of a system that can be executed immediately, so as to simulate the behavior of the system to be designed. It has a specified object quickly prototyped, then can be checked for design consistency, system verification and performance simulation. Moreover, an executable specification is able to accomplish more than an ordinary simulation does. A common simulating model promises only to respond to particular aspects of the behavior of the system being simulated, without making any claims to represent the system in a comprehensible and instance-independent way, whereas an executable specification must support those claims as well as reproduce behaviors. Thus, it makes a true performance evaluation as early as the design's inception.

The word "operational" is used to name such specifications for an emphasis on building an operating model of the object system functioning in its environment [Zav80, 82a, 83]. The

definition of "operational" originates from the notion of the operational semantics of programming languages [Weg72] in which the meaning of a language construct is defined in terms of an observable sequence of state transformations resulting from the execution of some abstract "computer" interpreting the construct. Zave [Zav82a, 83] applies the "operational" approach to techniques used for modeling an executable specification, so as to find internal inconsistency and simulate the behavior of the specified system.

In most modern CHDLs, "operational" is widely accepted as an algorithmic or programmatic way of description [Das84a, 89b, WilsD89] without emphasis on its ability to involve an early system performance evaluation. Two words: "operational" and "executable" will be used alternately in this paper with closer ties to Zave's definition and the original meaning of Wegner [Weg72], i.e., through the execution of operational specification, we reproduce the behavior of the specified object and gather performance measurements directly from the trace of computation. It is more substantial because it makes an early system evaluation and analysis truly possible and practical. An executable specification is superior to an ordinary simulation in its ability to describe the entire system in a "comprehensible" and "instantiation-independent" way [Bou91, HabJ90, Zav82a, ZavS86]; furthermore the regular simulation itself is easily trapped into several special instantiations and overshoots the generality and flexibility of the system design.

Although the directly executable approach is largely untried in the specification of computer systems, it does offer many potential advantages [Zav83], including: "(1) fully formal, and therefore rigorous, unambiguous specifications subject to static checks for many kinds of internal inconsistency; (2) rapid prototyping as an inherent part of the development process; an operating model to reproduce behaviors and gather generic performance measurements; (4) freedom from the implementation errors; (5) improved automatability, maintainability, and accountability; (6) enhanced communicating vehicles with customers and within design staff".

The first advantage is a benefit obtainable directly from formal specification, while the rest are

produced from executability. It is because of executability, an automatic interpreting of specification as in the case of a simulation, that this process can be used to demonstrate and test the design object, or even be released as a prototype. This improves user participation in validating proposed system behaviors. It is due to executability that a specification can be debugged and checked for internal inconsistency, and then evaluated by the analysts who wrote it, insuring early validation and accountability. That is another reason why we credit PAISLey with accommodating system specification, though it is generally a multi-level specification language.

The operational specification of other systems has been fairly well discussed in software engineering [Bac78, BelB77, BergZ86, Weg72, Zav91, ZavS86, ZavY81]. Nevertheless, when this research was begun, not much work had yet been done in this field regarding its application to computer hardware description and computer system performance evaluation. The executability and the direct performance evaluation obtained by executing the specified objects had not received deserved attention or serious study at that time, due to the fact that the most hardware description languages had it split into two stages and converted the specification to the simulating model by using special internal or external tools. That easily digresses from the original specification. At present, several papers propose various executable specification languages as computer hardware description languages by applying temporal logic - such as Tempura [Hal87], FCP [DotA90], TRIO [CoeM91, GheM90], etc. This will be discussed in section 2.7 when the PAISLey language will be compared with other CHDLs.

2.3. System Specifications and Abstract Levels

The word "system" is carefully used here, instead of "computer hardware", for the purpose of emphasizing that the specification is for the complete computer system, not merely a hardware description. As we all know now, modern computer architecture is no longer simply a pure

hardware problem. The impact of computers and their system management and development programs are bidirectional and profound. This is due to the fact that a computer normally works under a system control program like an operating system or other dedicated system software. The facts that the CPU must mutually support operating systems such as memory management, interrupt processing, ...; that the instruction set of contemporary RISC machines [PatS81, 82] is more compiler-oriented ..., and; that the very long instruction word parallel architecture [FisR91] depends on VLIW-style object codes produced by a optimized compiler are all well-known examples. There exists a wide spectrum of interaction between hardware and software systems. Such activities, as we predict, will increase even more in forthcoming computer architectures. As a result, the specification of those interactivities between hardware and its control and developing software are influential and crucial as is the specification of hardware. Nevertheless, this aspect is frequently ignored in previous work because it could be easily deceptive in that HDLs are used only for hardware descriptions.

Using PAISLey, it is not difficult to specify those interactions. A computer can be described as a digital component embedded in an operating environment, say an operating system, and then we may be able to express the behaviors of those activities with the internal nature of the operating system left undefined. Or, we could specify several autonomous processes which represent external characteristics of the operating system that are receivable to the computer system, such as virtual address mapping, trap examining and handling, etc. Eventually, we can arrive at decisions for the compromise between hardware and software during early performance simulation.

The design of a complex computer system is often viewed as a hierarchical system where each higher level is an abstraction of the lower. The purpose of this hierarchical view is an attempt to divide the computer system into smaller parts that can be more easily managed and specified. The design staff at each abstraction level need not care about the additional details irrelevant to them, thus, dramatically reducing the complexity of design. Generally, the partition of different

levels depends on the components recognized as primitive or indivisible. However, in practice, the actual division depends on current technology, available tools, and the preference of the individual designers. In computer literature, there exists a variety representative hierarchical views [Bel71, Bae80, Das89a, Hay88, Mye82]. Adapted partially and refined from Dasgupta [Das89a, Hab91], we present our view of the abstract levels of computer architectural design in Figure 2.1, based on modern computer technology and system specification.

Endo-architecture is concerned with the functional capabilities and the interconnection of the physical components of the machine, and how information flows between these components. Micro-architecture is the view from the micro-programmer. Micro-architecture as a refinement of endo-architecture and separate level becomes especially perceivable when the system has programmable control stores or logical arrays. In addition, micro-programming is still used in CISC and VLIW style parallel machines because of its modularity and retargetability. Nano-architecture is a register-transfer level that mainly concerns the micro-operations and micro-signals seen by register and logical level designer.

It is worthwhile to discuss the top two levels: system specification and high level hardware design which are the layers usually viewed by an assembler programmer and compiler developer. It is a sketch of global computer system structures and a collection of all external observable characteristics. The level of system specification, which we claim is extraordinarily well-suited for PAISLey, has been drawn a half-level above that of high level hardware design, with the emphasis on that computer system level specification has to concentrate on those interactivities between computer hardware architecture and its environmental system software and it balances the overall behavioral and structural quandaries of an entire computer system, as well as describe hardware. System level design may specify and address more generic behaviors because we only consider the effects of system software instead of its internal structure and we also do not have implementation structure details at the system design level. Making decisions at the system specification level is a critical and profound task in early computer system design that might

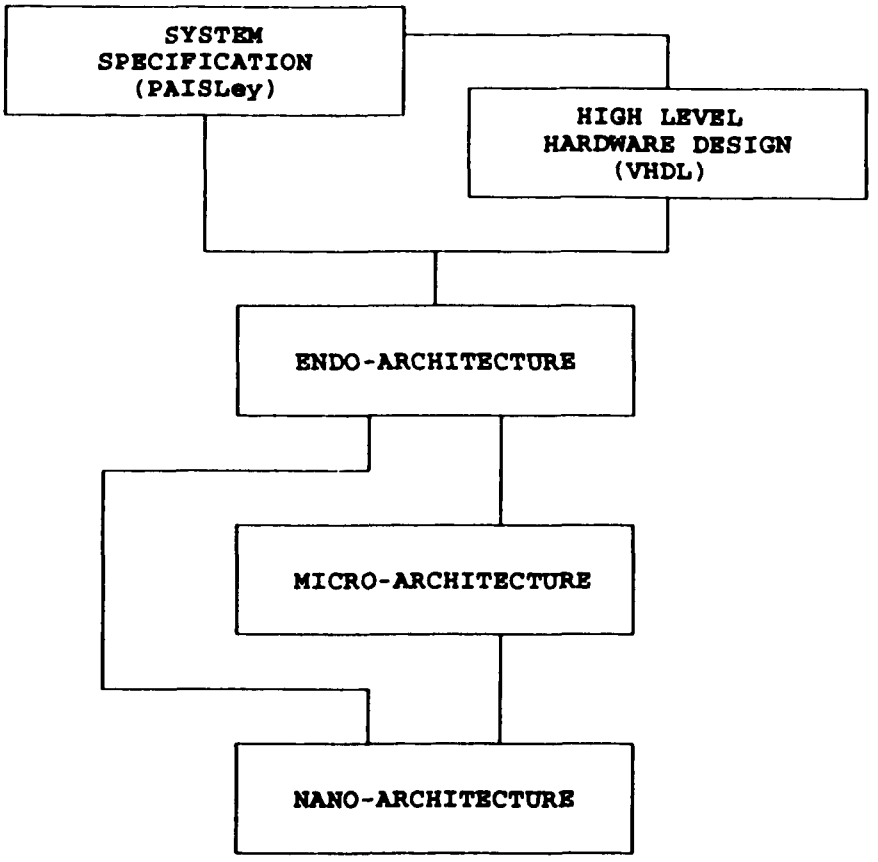

Figure 2.1 Abstract Levels of Computer Architecture

determine its whole success in future development. Unfortunately, as we have observed, most divisions of computer architecture levels neglect or, at least, do not place enough consideration, on those activities in a system level design. By contrast, high-level hardware design or description is oriented more or less to the implementation techniques and target hardware structures after compromises have been made between system software and computer hardware during the system specification. Though it is true that a large part of system specification may overlap that of high level hardware design, the latter normally describes only structural connections and functions of hardware components at the processor level with limited ability to make a trade-off from the system software management environment. That is why the layer of system specification is drawn

a half level higher. We can even by-pass the high level hardware design if we well contemplate the hardware structure when the overall system behavior is being specified as we do by using PAISLey.

We would like to point out that the boundaries between the layers are not so clear-cut. No matter how suitably the partition is made, it is quite common to encounter specifications which contain the abstract modules from more than one level, essentially in the top three levels. Overlaps at the those boundaries are inescapable. As a result, modern CHDLs are usually required to be multi-level descriptive. It makes PAISLey even more useful as it is indeed a versatile and multi-level specification language. The tolerance of incompleteness and the utilization of functional properties allow stepwise refinement and cross-level interaction to be specified smoothly and transparently.

Though it is not important to choose which division for specification, the capability of specifying multi-level abstraction is a fundamental requirement for modern CHDLs. It is due to the following:

(1) Stepwise refinement is a helpful property of the specification languages. In using the same language as a top-down design medium, a specification can be successively refined to a more target-close detailed representation, so the specifying complexity is largely decreased and instance details can and should be left to lower implementing level.

(2) The boundaries between the levels are not clear-cut. No matter how subtle the division made, it is quite common to encounter specifications which include the abstract modules from more than one level. Therefore, the one-level ability will constrain the use of the specification language.

(3) Finally, learning one language is always easier and more convenient than learning

several languages for different abstract levels.

The capability of PAISLey as a multi-level specification language will be studied extensively in Section 3.2. We argue that PAISLey adequately sustains those fundamental requirements and fully accomplishes a rule as a computer system description language.

2.4. Applicative (Functional) Languages

Applicative specifications are those written in an applicative language. An applicative, or functional, language is the language based on side-effect-free evaluation of expressions formed from constants, formal parameters, and functional operators ("combined forms" for function, such as composition). Well-known examples of applicative languages are the lambda calculus, pure LISP, and the functional programming system [Bac78].

The most distinctive method of viewing an applicative specification is as functions which are also called mappings or processes, and the most remarkable operation or computation is the application of a function to a tuple of bound parameters. More descriptively, specifications using an applicative language to define behaviors of computer systems in terms of a function (mapping) from a set of system inputs to a set of system outputs, without revealing how the system performs the function. Thus, an applicative specification does not concern itself with the method used to implement the specified function, rather, it treats the function or system as a "black box," the behavior of which is defined only in terms of input and output relationships. Therefore, an applicative specification can be easily made fully free from the implementations.

Applicative specification is currently receiving a great deal of favorable attention, because of its numerous theoretical and practical advantages [Bac78, Das89a, 89b, Sok91, Zav82a], most of which are well suited to the goals of system specification. Virtues of an applicative specification

are summarized in the following text:

High order logic - Applicative specification allows a very high degree of logic programming, that is, functions themselves can be treated as parameters and passed into other functions. It comes much more readily than imperative languages because of its logic nature. High order logic treats variables and functions uniformly, in other words, variables and functions can, interchangeably, be bound to the same formal parameter. This property greatly enforces its expressive capability and makes for a natural top-down refining.

Power of abstraction - The most important property of the functional specification is its tremendous powers of abstraction. In the stepwise refining design, an applicative specification can easily identify distinct modules and create the abstraction for their models. That is because logic expressions specify merely the input and output relationship without any concern with inner structures. Applicative programming embodies this abstraction by using each function as a pure "black box".

Ignorance of non-interested aspects - There are situations in which we actually do not know the internal working of an abstract module, nor do we particularly care. In order to specify the interactions with an operating system, for instance, we do not care what the inner world of the operating system may be, but only concentrate our attention on input from and output to the processor. The "black box" nature of the applicative languages greatly facilitates such negligence of the inner workings of the unknown or uninteresting world.

Freedom from implemention - Since the functions or entities are considered as black boxes, only their input and output relations are specified, so an applicative specification is able to be fully independent from instantiations. In addition, an applicative language

specifies only what one should do, but not how one can do. Therefore, it needs not any implementing steps. This is a precious property for the system level design. Most imperative description languages do not have this ability because they normally requires the specification for how to do.

Formal manipulation - It is clear that an applicative specification is well suited to static analysis and verification, due to the rigorous and formal descriptions using the high order logic expressions. A traditional logic is well known for its ability of inference, consistency derivation, synthesis and verification.

Arguably human friendly - Due to the nature to describe what to do rather than how to do, a writer might feel an applicative style is more "friendly" than an imperative one. The former lets him/her focus on what formal relationships and objects occur in the design problem and what relationships are true about the desired solution, while relieving him of the concern over what should be done by computer processing. Additionally, the elimination of descriptive computer steps and higher order logic mathematics regularly make an applicative specification much shorter and simpler than the corresponding imperative one. Nonetheless, most experienced conventional programmers think it is a "rebel notion" and feel uncomfortable with it since an applicative language is truly less conventional and computer friendly.

Potential for efficient execution - When it first emerged a couple of years ago, an applicative approach did not receive its deserved recognition because of its decreased time and space efficiency as compared to conventional imperative languages. However, thanks to the rapid development of computer technology and wide application of parallel and distributed computing, this negative point now turns to be a most positive merit as it is characterized by a set of non-sequential functions. Many papers in software engineering [Bac78, Smo82, ZavS86] have discussed the great potential of applicative

programming for efficient execution. However, it is beyond the range of this work.

Though VHDL claims that it can produce functional descriptions, it is basically an imperative or procedural language, not an applicative one. Thus, it is unable to offer a completely instant-free system specification as an applicative approach does. By the time we began this research, few applicative languages were seriously used in computer hardware description language. Currently, more and more people see the limitations of CHDLs in specifying the intention of system design, and are starting to try different styles, particularly, temporal logic, and various refinements of applicative languages such as dialects of Prolog.

Before finishing this section, I would like to add some literary remarks. While most people think that the applicative and functional are the same adjective [Ten76] in terms of semantics of languages, in a general sense and an application level, some experts do draw a distinction at a finer classification [Smo82]. Functional programming is distinguished from applicative programming by its elimination of the need for identifier-binding construction, that is normally needed by applicative programming. PAISLey is a purely applicative language. Through this entire thesis, we make no such distinction by adopting an application attitude.

2.5. Tolerance of Incompleteness

A specification must tolerate incompleteness in addition to executability, in the sense allowing some missing definitions for data types, functions and modules. There are too many reasons for having some entities unspecified, especially at the system design level. One is that we may not be interested in the inner nature of several objects or we may just know its external functional properties, but do not know its internal workings. It is unnecessary and maybe impossible to specify those inner structures. Therefore, a specification, by any case in a system design level, should be executable, or usable in non-executable specification, even when it is incomplete,

because there are too many undetermined phases and too many interactions from the worlds of software management system, users, etc., which are commonly unnecessary or unknown for specification at the system level design. Sometimes, it is mere verbosity and excess to specify lengthy well-known entities such as widely-accepted standards and well-understood common-sense.

The word "incompleteness" deserves special explanation here, as we have herein used it for two different meanings. First, as we asserted in Section 2.1 that a specification must be complete, with respect to the fact that the final specification must enumerate all the requirements of the target systems without under-constraining. It stands for a faithful representation. The meaning used in this section allows for missing several definitions for some uninteresting parts or some parts outside the scope of the designed environment. Those parts does not influence any behaviors of the system being designed. Even in the last completed specification, the modules and data types reflecting those parts may be and should be left as undefined because its irrelevance to our target.

On the other hand, tolerance of incompleteness reinforces design flexibility. It contains two aspects. First, it helps the decision deferring and, secondly, it facilitates retargeting the system. For instance, from time to time, we wish to defer a decision to a single function, but want to test the part of the specification before adding the others. Then, the functions representing the deferred decisions can be left undefined or inhibited. Variation of the results coming from the both specifications with and without missing functions show the exclusive effect of those functions and assist designers in making a conclusion. Another very useful and helpful benefit from tolerance of incompleteness comes from debugging and testing. A useful rule-of-thumb says to do the work in three stages: first specify all the module skeletons and test each of them in isolation; second, add the communication paths and test the global architecture of the system, and; third, specify the abstract data types and test the complete specification. This strategy offers the quickest route to a validated global architecture, which is usually the critical issue and always the most difficult thing to change. Obviously, the isolation of modules is the intuitive need for

incompleteness. The other reason for leaving out part of a specification is that some part of the system may be well-understood or already implemented, hence not worth the effort of specifying it.

PAISLey tolerates this incompleteness, and all the undefined mappings are evaluated by arbitrarily choosing a value from its range or read interactively as an input from the designer or tester as its argument. All the undefined data sets will be considered as containing any data values. Almost all the other CHDLs, so far as we know, do not treat incompleteness as effectively as PAISLey does. Some do not even allow missing the definition of any single term appearing in the specification. This causes, very often, difficulty in specifying the interactions from or between unknown modules; the lengthy documents and superfluous redundancy for well known parts.

2.6. Timing Constraints and Early Performance Evaluation

Performance constraints are parts of the requirements of computer systems. Performance requirements are often expressed as the timing constraints on the successive evaluation in an applicative specification in order to describe real-time components. Thus, besides the precise specification of the functional behaviors, the timing constraints must be attached to complete an executable specification, which is normally used as a performance simulation paradigm, to meet the performance requirements. A timing constraint refers to the evaluation time of the functions in the PAISLey applicative languages. The evaluation time is frequently a random variable, and any information about its distribution, such as lower and upper bounds, mean or a distribution itself, can be prescribed by it. More generally, the sequence of evaluation of the function over the lifetime of the computer system could be associated with a stochastic process, so that the time of each evaluation would be a separate random variable. The stochastic processes applicable to the time constraints of PAISLey include constant, uniform, normal, exponential and hyper-exponential distributions.

Since the execution of a specification in PAISLey is automatically a performance simulation, an executable specification will be able to provide an early performance simulation whose results can be used in both static analysis and automatic coherence checking. Obviously, as indicated by Zave [Zav87b], "simulating the performance of a specification cannot have the same meaning and purpose as simulating the performance of an implementation. Performance simulation on a PAISLey specification, as an early performance simulation, has two possible uses: (1) to help establish the global consistency of the local timing constraints written in the specification, and (2) to establish global performance properties arising from the process-level timing constraints, for the purpose of validating them against informal requirements".

In addition to explicitly appending time attributes to its mapping functions, PAISLey also provides several global mechanisms to control the length of successor mapping steps and the time of function evaluation. These methods encompass setting the basic time units, revising the scale factors which adjust the relative rate of individual process, altering minimum overhead time, etc. All of these make the specification of timing property and performance requirements very flexible and versatile, and eliminate the deficiency of conventional logic, which traditionally has no timing features.

It is very important to know the timing constraints that result from performance are consistent because the consistency of timing constraints in a PAISLey specification manifests the feasibility of constructing the specified system with the required performance. The PAISLey interpreter automatically checks the local consistency within a process and performs partially a consistent check between processes. The complete global check has to be done by analyzing the trace results.

Most CHDLs provide time constraints, since a computer itself is a real time component. However, nearly all of them attribute time constraints as time delays to individual implementing entities and lack global time constraints, which make CHDLs more like the normal instanced simulation

languages rather than the system specification tools. PAISLey associates those time attributes to its executable applicative specification. Thus, performance requirements are also specified functionally and executably. Those timing constraints are attached as implementation-independent and can be checked for system inconsistency. More examples will be seen in Section 3.

2.7. Comparison of PAISLey to Other CHDLs

We have declared that PAISLey satisfies all fundamental requirements of computer system specification as mentioned in the previous sections. We choose the PAISLey language to make an attempt at fulfilling the gap between system specification and early synthesis-oriented behavioral simulation modelling. By using applicative high order logic and a proven executability, PAISLey specification is capable of providing a formal, operational, and implementation-independent behavioral validating and simulating model for an entire computer system design. Compared with most of current computer hardware description languages, PAISLey has, in addition to its high order logic formalism and fully-operationality, the following salient features:

Simplicity - The simplicity of PAISLey manifests itself in two manners. First, it provides a concise specification which preserves the most intuitive concepts from the original synthesizing intent. Contrasted to most CHDLs, PAISLey specification usually is much shorter. Second, consisting of only five different types of statements, the PAISLey language itself is a relatively simple language. With little rudimentary knowledge of logic functions and relations, one can very easily gain proficiency in PAISLey.

Power of abstraction - PAISLey's most powerful expressiveness of abstraction is a direct result of applicative high order logic. Since functions can be expressed naturally as parameters of other functions based on uniformly encapsulated computation, a pure

relationship in truly "black box" style can be faithfully specified without referring to any technologically-independent inner structures. Refinement of individual functions or processes can be carried out at any particular time with no effects and impairments to others. First order logic used to inhibit the application of functions to parameters in functional tuples, and procedural languages pass functions as parameters in an unnatural and unpleasantly limited way. That hampers the expressive power of abstraction in those languages.

Abundance of time attributes - In order to solve time deficiency in this traditional logic, time constraints can be attached to any functions in PAISLey specifications. They can be a constant, upper-bound, lower-bound, or a random variable with a uniform, normal, exponential, or hyper-exponential distribution, which refers to the evaluation time of the functions. All time constraints are interpreted as simulation time when they are successful during execution. Failures of time constraints report the system inconsistency and design error. Even better than the time delay specified in most CHDLs, time constrains can be connected to a very high-level function and a global implementation-independent time specification and can be enforced with no requirement to describe any time delaying details, which are normally technologically-, that is, implement-dependent.

Tolerance of incompleteness - Two mechanisms are provided by the PAISley language to tolerate incompleteness. Note "completeness" here means closure of definitions and declarations. One is the permission of incompletely specified events and processes: a complex subsystem or event may be left partially or completely unspecified at any moment and the system needs only know its existence or the distributively occurring time, the completeness of these unspecified events being fulfilled later during the refining stages or even entirely ignored if we are not interested in such. The other is the ability to inhibit the evaluation of individual functions with no need to modify the

specification. PAISLey treats the deactivated function as an undefined one and, instead of evaluation, picks up a value randomly or constantly from the range of the function when the function is encountered during the execution. It is especially valuable in debugging in that it allows the isolation of some functions from others or tests the exclusive effects of certain specified objects.

Maximum degree of parallelism - Expression of parallelism in PAISLey comes from the merging of asynchronous processes and synchronous functions. Asynchronous computation is specified by separated processes which are executed concurrently. Synchronous functions are placed as arguments in one tuple and also evaluated in parallel. It is said to be maximal by the fact that the synchronization is only required for those functions in the same processes at the endpoints of process cycles. Procedural CHDLs, like VHDL, fail to capture this parallelism because the textual orders in those descriptions impose the executing order. Parallelism in those languages has to be expressed in an artificial way.

Interface to other languages - Functions undefined in PAISLey specification can be defined by other high level languages such as C. Functions defined in C are treated as the same uniform functions as defined in PAISLey. All are run concurrently and transparently. Many reasons exist for having a high-level specification language open a window to others, for instance: (1) leaving awkward and useless implementation details to a lower level language can make system specification more concise and understandable; (2) making it possible to use utilities in the existing library of a languages, especially input and output routines, whereas most high-level specification languages suffer from lack of the flexible reading and writing capabilities leading to a compacted and concentrated specification; (3) adding a semantics for a multi-paradigm interaction. A multi-paradigm approach is being experimented upon nowadays to gain mutual benefits from each other's best-suited classes of problems.

CHDLs can be mainly divided into two classifying groups. One employs an imperative and procedural programming style, of which VHDL is a typical representation. The other turns to the logic programming style like PAISLey, Prolog and temporal logic, etc. Standing in different classifications, it is actually unfair or at least improper to compare VHDL, in an imperative and procedural nature, to the powerful abstract expressiveness of applicative high order logic languages such as PAISLey, with respect to system specification. However, since VHDL is now adopted by IEEE as a standard CHDL, though it has not gained wide acceptance in the hardware community [CoeM91], proposing or selecting any other language will certainly raise the issue of VHDL's shortcomings. VHDL is well-known for its less expressive capacities in instance-free syntheses and system specifications and register-transfer level description [Bou91, CoeM91, Goe92]. Regarding synthesis design and system specification, we think VHDL lacks the following required features [Agn91, NasS86, Goe92]: (1) VHDL is basically a event-driven simulation language for medium level structural description. It is difficult to capture designers' intent using VHDL and fails to provide an implementing-independent system level specification. (2) VHDL has no facilities for data transfer in and out of a design entity. Initializing the data for each design entity is very time-consuming. However, as the design entity is expressed as a process in PAISLey, the data can be transferred freely by the primitive functions called "exchange functions" between the asynchronous processes. (3) VHDL prohibits the use of global variables. All data going to a design entity must pass through the interface definition. This design decision eliminates the known language problems due to side effects and improper use of global variables. Conversely, it makes the utilization of globally-declared system variables difficult. (4) VHDL cannot suspend sequential statement execution during the simulation, and then continue based on the passage of time or occurrence of some condition, while the execution (the automatic simulation) of PAISLey specification can be suspended at any point and resume right from this suspended point. (5) VHDL sacrifices terseness for documentation and readability, that may be the common fault of the DOD projects such as Ada, due to the large scale search, a giant and distributed working group and huge investment of money. PAISLey specifications are far more concise. (6) The complexity of descriptive methods and various modules is the strength of

VHDL, but it is also the defect as it makes learning VHDL difficult and time-consuming and it costs for design staff. (7) VHDL does not support electric circuit description and gate level design well. (8) VHDL misses the analog simulation model. We would like to put one comment here. Although, the critique of VHDL is based on IEEE 87 version, an updated version, IEEE 92, a new VHDL standard is currently being developed; doubtless some of the missing features will certainly be found in this new version as well.

Extension of VHDL is appealing and proposed by many papers [Agn91, BorW91, Goe92, Nol92] for years. However, the weakness of VHDL in system specification resides in its nature of the procedural and imperative descriptive way which VHDL adopts mainly for event-driven simulation. Consequently, the extension of VHDL to enhance this aspect will inescapably insert some different descriptive approaches and modules to an already complicated system. Therefore, the new VHDL will still be unable to bring all those explicit virtues carried by the applicative logic programming. Recognizing this, several writers have pioneered other descriptive methods, especially in logic programming such as lambda-calculus, applicative high order logic and temporal logic. PAISLey, of course, is one of them. Among others, there exist Tempura [Hal87], FCP [DotA90], TRIO [CoeM91, GheM90], and so forth. It is unquestionable that logic is strong and expressive when dealing with abstraction, relationships and other static situations. However, traditional logic falters in dealing with time. Thus, a critical issue in logic type specification is how to specify time constraints to get a quantitative analysis during execution. One solution, as exhibited by PAISLey, introduces explicit time constraints into the functions to be defined, while the other, widely employed in temporal logic, generates the predicate logic by including the notion of an implicit temporal domain. Tempura is an executable subset of Interval Temporal Logic (ITL). In order to allow for executability, Tempura keeps several of the more hindering features from imperative languages by using assignable variables, explicit notion of state, strict sequential execution, which ignore most of valuable features of logic. Flat Concurrent Prolog (FCP), a member of Prolog family, has been claimed to satisfy all requirements for a hardware description. However, other people dispute that it may result in long programs which are not immediately

readable and are not clear in some cases. Besides, time constraints for time dependent entities in FCP are treated in a very artificial way which means time specification is indirect and always in discrete form. Although TRIO, another new temporal logic based language, states that it is able to deal with time constraints effectively and sufficiently, it still uses first-order logic. Therefore, it is unable to be as powerful as high order logic of applicative languages in expressing abstraction.

We have no intention of implying that PAISLey is perfect. Such a language does not realistically exists. During the process of specifying the WLAB cache systems, we found that PAISLey lacks an ability to deal with a batch mode of input data. The original PAISLey only accepts an interactive input, one value at a time, or picks up either a default or a random value in the predefined range. This is obviously a lesser ability in most performance simulation situations, since a large volume of input may need to be fed in from an external data file. For instance, in trace simulation as we touched upon in the performance evaluation of the WLAB cache systems, thousands of trace references needed to be read one after one when simulation is going on. However, it is not hard to overcome this shortcoming. We use a C process to get next references. C functions can be computed concurrently and uniformly in PAISLey specification. They appear in PAISLey specification as an undefined functions and the PAISLey interpreter will call them automatically and run them with other PAISLey processes in parallel. Writing a C routine to read input values is not a complex task. On the other hand, by leaving I/O processing specification out, let designers concentrate more on target system describing, which eventually leads to a more compacted and readable specification.

Another arguable point in PAISLey's favor is its readability because of its logic programming style. This question has long been the subject of debate starting with advent of lambda calculus languages, LISP and Prolog. The supporters contend that it is more natural and closer to human thoughts. The opponents consider it a rebel of conventional programming languages such as FORTRAN and Pascal creating difficulties for programmers already familiar with them. The

discussion of this is certainly beyond the context of this thesis. Our philosophy here is to choose a well-suited executable specification language to fulfill our mission of eliminating a gap between a high degree of abstraction and instance-free system level specification and an operational early performance simulation model. The applicative high order logic language, PAISley, is a satisfactory choice for this task.

3. PAISLEY SPECIFICATION MODELS

This section, serving as an illustration of the capabilities of PAISLey, exposes how PAISLey, being a computer hardware description language, is able to compose a succinct specification for computer systems. To recap, a specification in PAISLey is consistent, precise and implementation independent - in the sense that the specified model can be realized by a wide variety of mechanisms on a wide range of technology and resource configurations.

3.1. An Overview of PAISLey

PAISLey is a process-oriented, applicative, and interpretable specification language. By using applicative high order logic and a proven executability, a PAISLey specification is capable of providing a formal, operational, and implementation-independent behavioral validating and simulating model for computer system design. The most outstanding characteristics of the PAISLey language is that it implements the concise executable specification which matches, faithfully, the initial intuitive design, a property lacking in most current computer hardware description languages. We choose the PAISLey language to make an attempt at fulfilling the gap between system specification and early synthesis-oriented behavioral simulation modeling. Compared with most of the current computer hardware description languages, lead by VHDL [IEE87, ShaL85], PAISLey has, in addition to its high order logic formalism and full operationality, superiority in simplicity which means that it is a relatively simple language and provides one with a shorter and more concise specification. With a little primary knowledge of logic functions and relations, it is very easy to learn the PAISLey language.

PAISLey consists of only five different types of clauses or statements: set definition; system declaration; function (called mapping in the original manuals) declaration; function (mapping) definition; and time constraints. Statements are separated and terminated by semicolons and comments are enclosed in double quotes. The same statements as those recognized by the C preprocessor such as "#INCLUDE" and "#DEFINE" can also be used for programming convenience. Look at the example shown in Figure 3.1, which is a specification of a 50 MHz clock cycle. The four statements represent four different types of statements. The first one known as a set definition that defines a clock pulse level, 1 standing for high voltage level and 0 for low level. Statement 2 is a system declaration which declares a clock process. A system specified in PAISLey is composed of processes. Each process is an active computational entity within the system, and runs concurrently and asynchronously with respect to all other processes in the system. In Figure 3.1, only one process "clock-cycle" is declared. Statement 4, called a function declaration, specifies the domain and range of process "clock-cycle". Domain defines a set of the legal arguments which a function (called a "process" or "mapping" in PAISLey) can take while range defines the value that will be produced when the function is evaluated. Both domain and range are defined by the set definition statements and referred to by set names which must all be in capital letters, whereas all process and function names must be in lowercase letters. Statement 1 is a set definition that associates a set name "PULSE" to an enumerate set {0, 1}. Sets can be declared as one of six predefined intrinsic sets - INTEGER, REAL, STRING, BOOLEAN,

```
"A Specification of a 50MHz Clock Generator"

PULSE = {1, 0} ;                                    "1"

(clock-cycle[1]) ;                                  "2"

clock-cycle: PULSE --> PULSE ;                      "3"
clock-cycle: !lb 20.0 ns, ub 20.0 ns ;              "4"
```

Figure 3.1 Specification 1: A 50MHz Clock Generator

FILLER and ANY. The first four intrinsic sets are self-explained, FILLER contains a unique element "NULL" for a function which takes or returns no meaningful data, and ANY actually is a typeless set that allows any data structures. Sets can also be defined from users by means of enumeration or set operations such as union (|) and cartesian production (*).

In an applicative style, an object can be treated as a real "black box" by specifying its external characteristics with no need to reveal its internal structures. We need only specify the pulse that should be produced by a clock generator to synchronize, or drive, the other outside components, but do not concern ourselves with how the pulse is generated. Such an abstraction, frequently required in system design, can be very well dealt with by using traditional predicate logic. However, such traditional logic specification usually has difficulties handling time constraints. PAISLey resolves this quandary by introducing an explicit time constraint statement which can attach the evaluation interval directly to a specified function. Statement 4, in Figure 3.1, says that a clock pulse must be generated every 20 nanoseconds, i.e., a 50 MHz clock cycle, where "lb" is a short notation for lower bound and "ub" is for upper bound. The identical lower and upper bound value means the exact amount of time interval.

The operationality of PAISLey specification is based on two computational models [Zav84a]: asynchronous interacting concurrent processes and functional programming. That is, similar to LISP and Prolog, computation takes place as binding of parameters and evaluation of functions. The execution of a specification starts with the processes specified in system declaration. A process goes through a sequence of discrete states in a set space defined in a function declaration for the process, which is either 0 (lower voltage level), or 1 (high voltage level), as in our example. Processes are evaluated cyclicly by applying the current evaluation results as the input arguments to the next cycle. All the clauses are parallel and will be executed as soon as all they are triggered, i.e., when all the functional arguments are ready and time constraints are satisfied. The textual order in a file does not imply the execution order. Figure 3.2 displays a partial trace from the execution of this clock generator specification. It clearly demonstrates that this

specification correctly describes the design intent -- generating one pulse at every 20 nanoseconds. The correctness shown by tracing outputs also verifies the consistency and feasibility of the objects being designed.

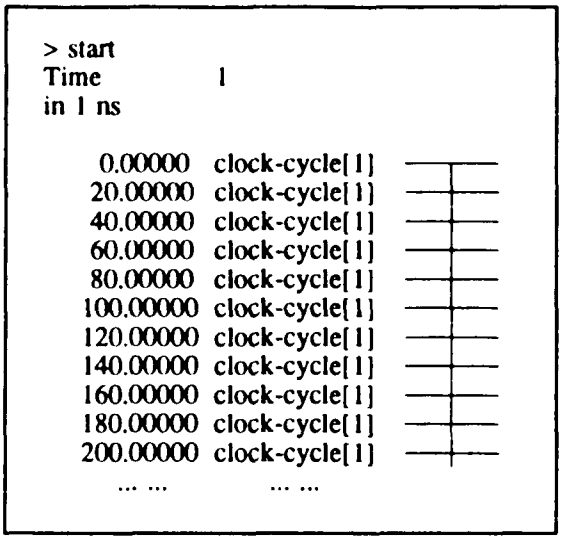

Figure 3.2 Trace of the Execution of Specification 1

PAISLey employs applicative high order logic, therefore, it has a tremendous power of abstraction. A design specification can start at a very high abstract level and then can be refined in a stepwise fashion to very detailed lower level design. That is because high order logic allows that functions themselves be treated as arguments and the result of its evaluations can be passed to other functions. Logic programming, to a high degree, deals with such argument binding and parallel execution much more naturally and easily than imperative languages. For instance, suppose we want to disclose more details of a clock generator and would like to know the up- and down-edge of each clock cycle. Then, specification 1 can be refined as the one shown in Figure 3.3. A function definition for "clock-cycle" (statement 5) is now added. Two more functions "up-edge" and "down-edge" are declared with "down-edge" used as an argument of "up-edge" to ensure a sequential effect. The completion of "down-edge" will trigger the start of "up-edge", and afterwards the terminus of "up-edge" concludes one clock cycle and initiates another cycle.

Clause 5 of Specification 2 introduces the last type of the PAISLey statement which we have yet to encounter - function definition. It defines functions and processes in terms of mapping expressions, and may use formal parameters to do so. A mapping expression may consist of just values, another mapping or the composition of the other mappings. The square brackets on the left-hand side enclose a tuple of formal parameters, say "*level*" in this example, used to refer to

```
"A Refined Specification of a 50MHz Clock Generator"

PULSE = {1, 0} ;                                                    "1"

(clock-cycle[1]) ;                                                  "2"

clock-cycle: PULSE --> PULSE ;                                      "3"
clock-cycle: !lb 20.0 ns, ub 20.0 ns ;                              "4"
clock-cycle[level] = up-edge[down-edge[level]] ;                    "5"

up-edge: {0} --> {1} ;                                              "6"
up-edge: !lb 10.0 ns, ub 10.0 ns ;                                  "7"

down-edge: {1} --> {0} ;                                            "8"
down-edge: !lb 10.0 ns, ub 10.0 ns ;                                "9"
```

Figure 3.3 Specification 2: Refinement of Specification 1

the current argument of the mapping. Formal parameter names are spelled with all lower-case letters like function and process names. The right-hand side of function definition is the mapping expression that returns the final resulting value by binding the formal argument and evaluation. A mapping expression is the generation of a mathematical function, that maps an argument from the domain to some values of the range (also called results) and differs from a mathematical function only in that the value need not be uniquely determined by the argument. Mappings are at the heart of PAISLey because it concentrates solely on the evaluation of mappings during the execution of a PAISLey specification.

Note that no function definition is specified for "clock-cycle" in specification 1, and the definitions of functions "up-edge" and "down-edge" in the refining specification of Figure 3.2 are missed. Why do we have to be bothered by these definitions since we are not interested in how the edges are generated internally? PAISLey specification will still be operational and usable with these functions undefined. Here is this big advantage of PAISLey, previously referred to as the

tolerance of incompleteness. Often, there are too many cases we may wish to bypass: non-interesting entities; or, save our energy on well-known parts; by-pass unknown world; delay tough decision to a later stage, and so forth. Tolerating those missed definitions and/or definitions equips the system specification with great flexibility and testability. During the evaluation, the PAISLey interpreter, acting like a modeling simulator, chooses a value from the range of a function whose definition is missing. The value selection can be implemented by the default, random number or interactive input. If the range of the function is missing too, then an arbitrary value, or interactively input value, will be chosen.

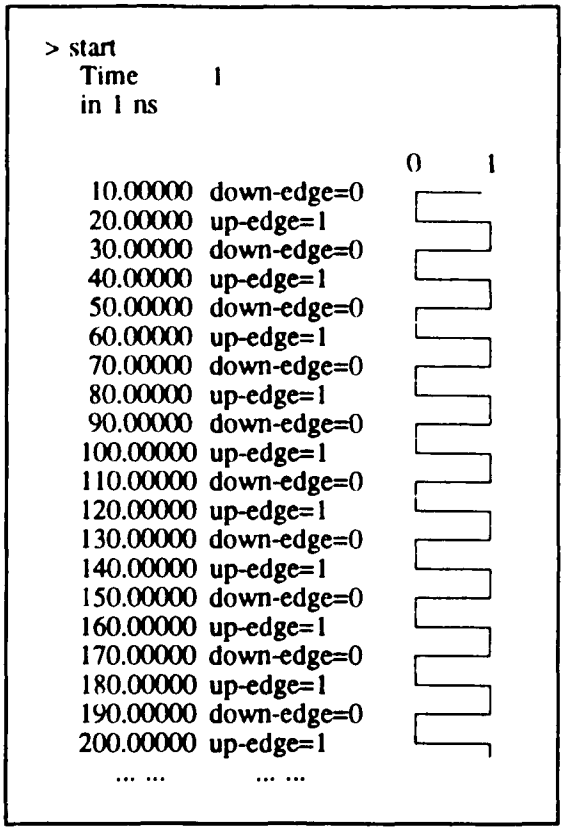

Figure 3.4 Trace of the Execution of Specification 2

However, for the purpose of checking internal system consistency, we encourage the declaration of function domains and ranges explicitly as long as we can. It helps the system's integrity. The trace of execution of the refined specification is given in Figure 3.4. Since there is only one element in both domains and ranges of function "up-edge" and "down-edge", any range value selection method, or even completion of the function definition, should not affect the simulating results. In those situations, we should make specification as simple as possible and add no artificial or technological constraints. Figure 3.2 lists the initializing points of the function evaluation that are denoted by the square bracket of argument, but Figure 3.4 reports the terminating points of the function evaluation that are symbolized by an equal sign followed by the argument with no square bracket.

Designed originally as an executable formal specification language for real-time systems, PAISLey offers preciseness, unambiguous description, completeness and executability. A variety of applications and a large number of the papers have been published concerning PAISLey since its inception [BerlZ87, Bru86, FilF84, Hab89, HabJ90, 92a, LippM88, McCZ89, Surl86, Zav89, ZavC83]. P. Zave summarized the actual results from those applications and specification paradigms in her recent paper [Zav91]. Many salient features of PAISLey make us believe that it is well suited for computer system modeling and hardware description. Several papers that maneuver PAISLey as a computer system specification language have been, or are about to be, published [HabJ92a, JinH92]. The rest of this section will demonstrate how well the PAISLey specification can deal with computer systems in a multi-architectural level. While, in the next section, we illustrate how we applied PAISLey specification to the WLAB hierarchical cache structure, our newly developed memory organization for multiprocessor systems.

3.2. Specification of Multi-Level Computer Systems Using PAISLey

Modern computer hardware description languages are required to be expressive of multi-levels, as is computer system specification. Therefore, we start the illustration at very low level: logical design or gate level, then proceed to higher levels, such as processor or system levels, as discussed in Section 1. We also address here a crucial issue of system specification: asynchronous interactions of two or more autonomous subsystems, a problem that fails or "embarrasses" many other CHDLs.

3.2.1. Specifications in Register Transfer and Logic Design Level

Though the specifications of computer systems normally go beyond the gate level design, in many cases small pieces of entities cannot be expressed at high-level, consequently specification at such low levels becomes mandatory. That is why a modern computer hardware description language

is always required to be capable of describing both combinational and sequential circuits at the register-transfer level.

Figure 3.5 shows a specification of a modulo-8 counter. The structure

j#1.. N < Delimiter Clauses >

is a repetition one in order to shorten the text and save typing energy for similar parallel components. It simply expands the clauses contained in the < ··· > N times, separating them by "Delimiter", and replacing every occurring j by the corresponding repeating number of each repetition. Accordingly, statement 3 in Figure 3.5 is equivalent to

COUNTER-STATE={0, 1, 2, 3, 4, 5, 6, 7} ;

```
"Specification: A Modulo-8 Counter"

#define   M    8                                                    "1"

(clock-cycle[1], counter[initial-counter-state]) ;                  "2"

COUNTER-STATE={ J#0..7 < , J > } ;                                  "3"
initial-counter-state: --> COUNTER-STATE ;                          "4"

clock-cycle: {0, 1} --> {0, 1} ;                                    "5"
clock-cycle: ! lb 20.0 ns, ub 20.0 ns ;                             "6"
clock-cycle[cp] = proj[(1, (cp, xr-impulse[cp]))] ;                 "7"

counter: COUNTER-STATE --> COUNTER-STATE ;                          "8"
counter[state] = count-next-clock[(state, x-impulse[Null])] ;       "9"

count-next-clock: COUNTER-STATE*{1} --> COUNTER-STATE ;             "10"
count-next-clock[(state, cp)] = mod[(sum[(state, 1)], M)] ;         "11"
```

Figure 3.5 Specification 3: A Modulo-8 Counter

For more instances, the following pairs are considered as equivalent:

```
#1.. 4 < * BINARY > ;
BINARY*BINARY*BINARY*BINARY ;

j#0.. 2 < ; test-j[state-j] > ;
test-0[state-0]; test-1[state-1]; test-2[state-2] ;
```

Two processes, "*clock-cycle*" and "*counter*," are declared in clause 2. Clauses 5, 6, 7 specify "*clock-cycle*" in the same way as in Figure 3.1, except that a function definition is explicitly defined in clause 7. Notice that *proj*[(i, (x_1, x_2, ..., x_i, ..., x_m))] is a PAISLey predefined function that returns the i-th elements in the second arguments, i.e., x_i. The interesting point here is the use of an intrinsic primitive - exchange function "*xr-impulse*". The purpose of this function is to export a clock pulse to other components, to the function "*counter*" in this example. When the counter perceives a clock pulse, i.e., another matching exchange function "*x-impulse*", in the mapping expression of function "*counter*" specified by clause 9, receives 1 sending from "*xr-impulse*", then the counter counts the clock and increments its internal state by 1 with modulo 8. The internal state transition of the counter is specified by function "*count-next-clock*" declared in clause 11. In PAISLey, any functions whose names start at *x-*, *xm-*, or *xr-* are called "exchange functions" which are used to establish the communication channels, or synchronizing points, between two or more autonomous subsystems or concurrently executed counterparts. They are particularly helpful when we want to deal with asynchronous or parallel processing. We provide more detailed discussion and applications of exchange functions in the following sections.

It is important, at the system level design, that we should be able to specify a general logical function required without involving any technical implementation. It must be a general function to accurately reflect the external relationships, not the specific circuits, like regular integrated gates or PLA to implement such a function. By using applicative logic, PAISLey conveniently realizes

this task. Figure 3.6 gives a specification for an arbitrary logical function with four variables, i.e., f(w,x,y,z), where "|" is the union operator of sets and * is the Cartesian product of sets. The mapping expressions 5.1 and 5.2 show the control structure of the PAISLey language as having the general outline as follows:

```
function-name[arg] = / condition-1:  mapping-1,
                       condition-2:  mapping-2,
                          ... ...       ... ...
                       condition-N:  mapping-N,
                       True:         mapping
                     / ;
```

Similar in operation to the *case* statement in Pascal, the conditions are evaluated one after the other in the text order. When one condition is true, the corresponding mapping expression will be evaluated and its resulting values will be returned as the value of the function-name specified in the left-hand side of the clause, while all the remaining conditions and mapping expression will

```
"A Logic Function f(w,x,y,z) "

#define   Nvar    4                                                    "1"
#define   Max     15                                                   "2"

f: INPUT-TUPLE --> BINARY | {Dont-Care} ;                              "3"
f[input] = f-mapping[(input, truth-value-f)] ;                         "4"

f-mapping[(input, (j#0.. Max < , (minterm-j, output-j) >))] =          "5"
          / j#0.. Max < , equal[(input, minterm-j)] : output-j > ,      "5.1"
            True                                    : Dont-Care         "5.2"
          / ;

BINARY = { 0, 1 } ;                                                    "6"
INPUT-TUPLE = #1.. Nvar < * BINARY > ;                                 "7"
```

Figure 3.6 Specification 4: A General Logic Function f(w, x, y, z)

be ignored. If none of these conditions is true, then the last mapping expression will be executed and its result will be used because condition *True* is always true. Accordingly, clause 5 of Figure 3.6 expresses that one proper output value will be generated when the input value matches one of the specified input combinations, for example, a minterm. The function is arbitrary so far because no truth values are furnished in Specification 4. To be specific, a truth table of the function can be supplied with the following definition:

```
truth-table-f = (   ((0,0,0,0), 1),
                    ((0,0,0,1), 0),
                        ... ...
                ) ;
```

Note that the truth table does not have to be completely specified. Any missing term will be interpreted as a "Don't Care" term because none of conditions "*equal*[(input, minterm-j)]" will be true and the last condition "True" will be triggered, thus the constant value "*Dont-Care*" will

```
TUPLE=J#1.. N < * BOOLEAN > ;

b-not: TUPLE --> TUPLE ;
b-and: TUPLE*TUPLE --> TUPLE ;
b-or: TUPLE*TUPLE --> TUPLE ;
b-xor: TUPLE*TUPLE --> TUPLE ;

b-not[(j#1.. N< , a-j >)] = (j#1.. N< , not[a-j] >) ;
b-and[((j#1.. N< , a-j >), (j#1.. N< , b-j >))] = (j#1.. N< , and[(a-j, b-j)] >) ;
b-or[((j#1.. N< , a-j >), (j#1.. N< , b-j >))] = (j#1.. N< , or[(a-j, b-j)] >) ;
b-xor[((j#1.. N< , a-j >), (j#1.. N< , b-j >))] = (j#1.. N< , ex-or[(a-j, b-j)] >) ;

b-not: ! lb 9.0 ns, ub 11.0 ns ;
b-and: ! lb 9.0 ns, ub 11.0 ns ;
b-or: ! lb 9.0 ns, ub 11.0 ns ;
b-xor: ! lb 9.0 ns, ub 11.0 ns ;
```

Figure 3.7 Bit-Wise Logical Operations

result. Observe that only the pure input and output logic relationship is specified here, adding maximum freedom to the implementation stage and obtaining a truly correct design performance. To be more expressively powerful, we can place more general terms such as (0, -, 1, -) in the definition of "*truth-table*".

In register-transfer level design, it will frequently be required to specify the bit-wise logical operations: not a difficulty task in PAISley, too. Look at Figure 3.7 where we define the bit-wise logical operations including "*b-not*," "*b-and*," "*b-or*," "*b-xor*," and observe that "*b-*" is placed at the beginning of all bit-wise logical functors in order to make a distinction from the PAISLey intrinsic non-bit-wise logic operators "*not*", "*and*" and "*or*". The latter accepts logical arguments and results in a new logical value, for instance, the value of "*not*[True]" is False and that of "*or*[(True, False)]" is True. However, all bit-wise operators receive n-ary logical vectors and return a new n-ary binary one in accordance with the operator for an arbitrary integer number, n. Thus, "*b-and*[((False, False, True, True), (False, True, False, True))]" returns (False, False, False, True) and "*b-not*[(False, False, True, True)]" results in (True, True, False, False). The last four clauses of Figure 3.7 specify the global delay time, a constraint that must be satisfied during design and implementation. Consequently, PAISLey is shown to be convenient for bit-wise operations of any size and endows an uncomplicated specification of register-transfer level design, which is usually a painstaking task by using VHDL, the leading imperative hardware description language.

```
#include"pdefines.h"
#include<stdio.h>

P_Valueptr  P_proj(), P_makevalue();

P_Valueptr  n_ary_or (tuple)
  P_Valueptrtuple;
{
  int  result=0, i=1;
  P_Valueptr  p;
  P_Atom  a;

  for ( ; ; i++)
    {
    p=P_proj(i, tuple);
    if (p==NULL) break;
    if (P->cvalue.boolean) result=1;
    }
  a.boolean=result;
  p=P_makevalue(BOOLEAN, a);
  return(p);
}
```

Figure 3.8 C Process of *n-ary-or*

In order to facilitate specification and shorten text length, we develop a library of C routines that provides two additional logical functions: "*n-ary-and*" and "*n-ary-or*" in terms of the suggestion of P. Zave [Zav87b]. They both perform an n-ary logical operation on a Boolean tuple of any size and produce one single logical result. For example, "*n-ary-or*[(True, False, True, False)]" yields a True result. PAISLey reserves the conventions of UNIX and provides a handy interface to C language. The object codes of compiled C processes are executed and interpreted transparently in the same way as PAISLey processes. Hence, users can call those C routines from the library as well as the predefined PAISLey functions, with no need to know how they are coded. Figure 3.8 shows function *n-ary-or*, where header "pdefines.h" furnishes the paisley interface to C language, and all the types and functions beginning with "*P_*" are processes provided by the PAISLey-C interface and defined in "pdefines.h." Details can be read from PAISLey User Documentation [Zav87a, b, c].

3.2.2. Specification in Higher Architectural Level

Since the partition of computer architecture is artificial and not clear-cut, we loosely use high-level to refer to any abstract level beyond register-transfer or logic design. It could be micro-architecture, macro-architecture, or any other refining abstract level. The significant point here is how we can specify, rather than how the division should be made or what this is. Consequently, the importance of PAISLey's capability to specify all abstract architectural levels is further demonstrated.

A computing cycle of a central processing unit, generally, includes two stages: instruction fetching and instruction execution. If these two stages are sequentially cyclic, we may specify a computer in the following system declaration:

(execution [fetching [(cpu-state, register-set, memory, cache, io-device, others)]]) ;

In accordance, the evaluation of the function *execution* will not start unless the evaluation of the function *fetching* is completed. The next fetching stage will begin right after the completion of evaluation of the current execution stage. The two stages will cycle one after one until it is forced to stop. It precisely specifies the behavior of computing cycles.

In order to increase speed, vector processing is adopted in almost all the modern computer systems. Among the various vector processing techniques, pipeline is the most popular one. As a result, these two computing stages are no longer pure sequential ones. They are "pipelined," in other words, they are working concurrently or simultaneously in different timing phases. Consistent with this, they now should be specified into two asynchronous processes to represent different pipeline stages[1]. The system declaration may look like clause 7 in Figure 3.9.

Figure 3.9 and 3.10, jointly, outline a specification for a pipeline computer structure of two stages - the instructions fetching and execution. Observe that "*fetch-stage*" and "*execute-stage*" are two autonomous processes that are synchronized by instruction passing, using the exchange functions at clauses 21 through 24. In the specification of the fetching stage, clause 11 reflects the fact that a computer usually fetches an instruction and sends it to the execution stage. Meanwhile, it increments its program counter (PC) during this stage and updates other relevant flags of states. However, the fetch stage will not read the next instruction until the execution stage accepts the current one. The execution states returns a "*Null*," when it receives the instruction, informing the fetch stage and letting it start the next fetching phase. Then, the execution stage starts the execution of the instruction just received by decoding the opcode which is normally the first byte of the instruction. The rest bytes of the instruction are considered as operand. Note that the two processes "*fetch-stage*" and "*execute-stage*" are not executed sequentially, rather parallel. While "*execute-stage*" is "decoding" and "running" the current instruction, "*fetch-stage*" is "fetching" the

[1] A modern computer architecture usually has more complicated pipeline stages. We adopt the simplest two stages here just for the purpose of illustration. Precisely, this kind of two stage pipeline is normally called instruction prefetching.

next instruction.

Briefly, some function definitions are omitted in Figure 3.9 and 3.10. We comment that PAISLey tolerates such incompleteness. By furnishing the real operate codes of an instruction set, i.e., giving the true opcode values to op-k, k=1, .., Num-of-Op, and feeding the objective program in machine codes, then the execution of specification 5 in Figure 3.9 and 3.10 will simulate the computing cycles of this target machine, although it is in a very high level.

PAISLey features also can be applied to microprogramming. Modified from the original version developed by S. Habib [Hab89], a PAISLey specification of microprogrammed architecture is presented in Figure 3.11. Referring to the previous section, a specification in PAISLey consists of asynchronous processes which are operating in parallel. In the case of a microprogrammed

```
"An Architecture With Two Pipeline Stages - Instruction Fetching And Execution"

#define   Instr_Length    4                                                    "1"
#define   Num_of_Op       128                                                  "2"
#define   Num_of_Word     4                                                    "3"
#design   Byte_Length     8                                                    "4"
#define   Word_Length     32                                                   "5"
#define   Capacity        65536                                                "6"

(fetch-stage[initial-fetch-cpu-state], execute-stage[initial-execute-cpu-state]) ;   "7"

initial-fetch-cpu-state:  --> CPU-STATE ;                                      "8"
initial-execute-cpu-state:  --> CPU-STATE ;                                    "9"

"Specification of the Fetch Stage"

fetch-stage: CPU-STATE --> CPU-STATE ;                                         "10"
fetch-stage[(pc, other-status)] =                                              "11"
            ( send-instr[fetch-instr[(pc, storage)]],
              pc-increment[pc],
              update-status[(other-status)]
            ) ;
```

Figure 3.9 Specification 5: Instruction Fetching and Execution Stages

```
pc-increment: ADDRESS-SPACE --> ADDRESS-SPACE ;                          "12"
pc-increment[pc] = mod[(sum[(pc, Instr_Length), Capacity)] ;             "13"

fetch-instr: ADDRESS-SPACE*STORAGE --> INSTRUCTION ;                     "14"
fetch-instr[(pc, memory)] = memory-read-cycle[(pc, memory)] ;            "15"

"Specification of Execution Stage"

execute-stage: CPU-STATE --> CPU-STATE ;                                 "16"
execute-stage[cpu-state] = execute[(cpu-state, receive-instr)] ;         "17"

execute:CPU-STATE*INSTRUCTION --> CPU-STATE ;                            "18"
execute[(cpu-state, (j#1.. Num_of_Word < , byte-j >))] =                 "19"
  /k#1.. Num_of_Op
     < , equal[(byte-1, op-k)]:  opcode-k[(cpu-state, (byte-2, byte-3, byte-4))] >,
   True:  exception-processing
  / ;

k#1.. Num_of_Op < ; opcode-k:  CPU-STATE*OPERAND --> CPU-STATE > ;       "20"

"Synchronizing Phase - Passing Instruction"

send-instr: INSTRUCTION --> FILLER ;                                     "21"
send-instr[instr]=x-instr[instr] ;                                       "22"

receive-instr: --> INSTRUCTION ;                                         "23"
receive-instr=xm-instr[Null] ;                                           "24"

"Data and Storage Definitions"

CPU-STATE = ADDRESS-SPACE*OTHER-STATUS ;                                 "25"
INSTRUCTION = WORD ;                                                     "26"
OPERAND = #2.. Num_of_Word < * BYTE > ;                                  "27"

STORAGE = #1.. Capacity < * BYTE > ;                                     "28"
ADDRESS-SPACE = { J#0.. Capacity < , J > } ;                             "39"

WORD = #1.. Num_of_Word < * BYTE > ;                                     "30"
BYTE = #1.. Byte_Length < * BINARY > ;                                   "31"
BINARY = {0, 1} ;                                                        "32"
```

Figure 3.10 Specification 5 (Continued): Instruction Fetching and Execution Stages

```
#define  Num_of_Micro_Opcodes    128                                    "1"

( micro-control-processor[init-micro-state],                            "2"
  macro-control-processor[init-macro-state] ) ;

micro-control-processor[(micro-addr, other-micro-states)] =             "3"
    proj[(1, (simulate-micro-machine[(micro-addr, other-micro-states)],
          offer-opcode[fetch-micro[(micro-addr, nxt-micro-instr)])] ;

macro-control-processor[macro-state] =                                  "4"
    proj[(1, (simulate-macro-machine[macro-state],
          opcode-decode[get-micro-opcode[Null]])] ;

opcode-decode[micro-instr] =                                            "5"
    (k#1.. Num_of_Micro_Opcodes < , recognize-k-op[proj[(k, micro-instr)]] >) ;

k#1.. Num_of_Micro_Opcodes                                              "6"
    < ; recognize-k-op[bit-value] =
                /equal[(bit-value, k)] : send-signal-for-k-op,
                 True : Null
                /
    > ;

offer-opcode[nxt-micro-instr] = x-opchan[opcode[nxt-micro-instr]] ;     "7"
get-micro-opcode = x-opchan[Null] ;                                     "8"
```

Figure 3.11 Specification 6: A Microprogrammed Architecture

architecture, the processes will consist of "micro-engine" and the "macro-engine". The micro-engine consists of the microprogrammed control store (usually in ROM) and the associated addressing mechanism. As the micro-engine progresses from state to state, the addressing mechanism fetches micro-instructions in terms of micro-address, decodes the command portion of the instruction and then issues micro-commands to the processor of the "macro-engine" which executes these micro-commands. The micro-engine then enters its next state by accessing the address portion of the micro-instruction and fetching the micro-instruction addressed therein. The macro-engine consists of the familiar central processing unit (CPU) which has received micro-commands from it. It carries out these micro-commands which are typically a set of micro-operations or a request to fetch a macro-instruction from the RAM. After the completion of the

current macro-instruction, a branch is taken to a micro-routine in the control portion (microprogrammed ROM) which issues a new set of micro-commands to the CPU to conduct the execution of the next macro-instruction. The macro-engine is a process whose state changes are reflected in the contents of the registers in the CPU and, if one includes the RAM as part of the macro-engine, the contents of the RAM also reflect state changes.

The processes associated with the micro-engine and the macro-engine are asynchronous with respect to memory-read-cycles and interrupts. When a micro-command is issued from the micro-engine to the macro-engine to perform a memory read, the micro-engine must await a reply from the macro-engine that the memory read is complete. This "handshaking" operation is just one of the asynchronous features of the interchange between the two engines. Another asynchronous operation occurs when interrupts from external devices, such as input/output devices, cause the macro-engine to service the external interrupt and halt the execution of commands from the micro-engine. We leave the detailed discussion of asynchronous specification to the next dedicated section.

In PAISLey, the way to specify processes is exemplified by the system declaration, clause 2 of Figure 3.11, for the micro-control-processor (the micro-engine) and the macro-control-processor (which is the macro-engine). The "*init-*-state*" for each process specifies the initial state value for its respective process. When the specification is executed using the interpreter tool provided, each process goes through a sequence of discrete states. Each process state is replaced by its successor state. These replacements encapsulate one cycle of the activity carried out by that process. In this way, each process is also termed a "successor mapping" in PAISLey.

In general terms, what the micro-control-process does is to access each of the micro-instructions in the ROM and then transmit the micro-op-code portion of that micro-instruction to the CPU of the macro-control-process. Relating this to the above notion of process state replacement, each time the micro-engine performs a new access to a micro-instruction and then passes the op-code

portion of that micro-instruction to the macro-engine, constitutes a process state cycle for the micro-engine. The process of cycling through each state of the micro-control-processor is carried out by performing an evaluation of mappings in terms of which this process is defined.

Clause 3, which defines the micro-control-processor, is interpreted as follows. The intrinsic projection function "*proj*" returns a first element or the resulting tuple (*simulate-* ..., etc.), which actually is the result of evaluation of the function "*simulate-micro-machine*". Function "*simulate-micro-machine*" is responsible for updating the micro-state, which is returned as a new argument for the successor mapping, and preparing the micro-address ("*micro-addr*") for next micro-instruction fetching. Function "*offer-opcode*" transfers the opcode of the micro-instruction to the macro-engine. It employs the PAISLey exclusively prominent synchronizing mechanism - exchange function "*x-opchan*". The next micro-instruction can be fed in either interactively or by reading from a data file via reading utilities we have developed.

The *get-micro-opcode* mapping in clause 4 of Figure 3.11 gets the opcode of a micro-instruction from "*micro-control-processor*", and then this micro-opcode is decoded by mapping function "*opcode-decode.*" At this point we have the contents of a particular micro-instruction as the argument of "*opcode-decode,*" and "*opcode-decode*" tests every particular micro-operation field and sends a signal to guide the respective micro-operation when the corresponding bit is set. "*Null,*" meaning no micro-operations here, will be returned if no action is required, as clearly specified in clause 6. Horizontal format is engaged in this specification; it can be easily modified to vertical format by testing the combinational value instead of every single bit. In an actual architecture what we have shown above as "*send-signal-for-k-op*" would constitute a command to CPU to perform a particular gating operation. For instance, the gating operation may be "*gate-contents-register-k-to-address*", etc.

In the specification of this microprogrammed architecture, we purposely do not include any set-definitions, and function declarations for domains and ranges in order to demonstrate that

PAISLey specification works more than suitably even without those statements. However, in most cases, we insist on clearly adding those definitions and declarations to expedite system consistency checking and document readability.

3.2.3. Specifications of Asynchronous Interactions

Any computer, no matter whether is universal or dedicated, works with peripherals, operating systems or some other form of control program. Many asynchronous interactions exist between these entities, such as virtual address mapping, interruption, spooled I/O operations, concurrent processing, etc. The impact of these interactions is significant and bidirectional to the evolution of the computer architecture. It may already be seen that almost all the modern computers have the hardware mechanisms to support memory management, some may even have the synchronous mechanisms to support concurrent processing. All these features have resulted from such mutual activities. Thus, the specification of those activities, and the capability of providing that specification as well the specification of hardware itself are important. Unfortunately, this aspect is frequently neglected in most of the past studies, partly because the lack of describing ability in some languages, and partly because the misleading nature of hardware description languages when used solely for hardware specification.

The most effective technique to support the specification of those asynchronous interactions is to consider the synchronizing mechanisms and communication tools in the specification language. The PAISLey language furnishes us with a set of primitives called exchange functions to accomplish this task. An exchange function is a primitive mapping which carries out two-way, point-to-point, mutually-asynchronized communication. It has one argument, which provides a value to be output, and returns a value which is obtained as input when a match occurs. A exchange function must start with one of *x-*, *xm-* or *xr-* prefixes, which are called "type attribute," and then follows a channel name that is a user defined identifier. For instance, the exchange function with type "*x-*" causes channel "*impulse*" named "*x-impulse*," the exchange function with

type "*xr-*" causes channel "*time*" named "*xr-time*," etc. Only the exchange functions on the same channel can match with each other. One channel matches another channel with the same channel name during execution. Each returns the argument of the other as evaluation results. However, one, and only one, pair of channels can be matched at any particular time. The matching channels exchange their arguments. As introduced in the beginning of this paper, a PAISLey specification is a set of asynchronous processes and each process is an autonomous one executed in parallel. By using exchange primitives, these independent processes can be synchronized and can communicate with one another via matching and argument exchange. We have already seen examples in Specification 3, 5 and 6 of Figures 3.6, and 3.9 through 3.11.

An activated exchange function that is ready to evaluate but without having found a matching counterpart channel yet, is said to be in a pending state. An x-type exchange function is able to match it with all types of pending functions, including itself, on the same channel. If no match function is pending when an x-type function is initiated, it will wait for a match. If there is more than one candidate for a match, the choice is non-deterministic but fair, meaning that each candidate has an equal opportunity without bias or discrimination. Thus, there is no possibility of lockout. An xm-type exchange function behaves exactly like x-type except that xm's cannot match themselves. The xm-type functions are commonly applied to a competition situation. An xr-type exchange function has a real time flavor, that does not wait to be matched. The xr-type exchange function simply returns its own argument immediately, or, after a certain time interval specified by a time constraint statement if no matching channel is pending when the xr-function is evaluated. Similar to xm-type functions, xr's cannot match with themselves either. The matching relationship is diagramed as an undirected graph in Figure 3.12. Here, loop stands for self-match. Since x-type channels are the only ones to be self-matched, both xm- and xr-types are said to be non-reflexive.

Thus, in the specification of a modulo-8 counter (Figure 3.5), *xr-impulse* (in Clause 7) and *x-impulse* (in Clause 9) can match each other when they are activated. Exchange function *x-impulse*

receives the clock pulse (*pc*) from *xm-impulse* of the "*clock-cycle*" process, then triggers the state progress of the counter. The exchange of arguments always takes place bidirectionally, though the other direction of exchange may not always be need. In the case of this modulo-8 counter, the intrinsic function, "*proj*," is employed to filter out the useless value "*Null*" returned by *xm-impulse*[cp]. The same rationale works for "*send-instr*" and "*receive-instr*" in the specification of the fetching and execution stages (Figure 3.10), and "*offer-opcode*" and "*get-micro-opcode*" in the specification of a microprogrammed architecture. These are the renamed exchange functions "*x-instr*," "*xr-instr*" and "*x-opchan*," i.e., the primitive is designated by a new regular function name with the same parameter as that of the original primitive, via a function definition. The renamed exchange function acts in the same way as its original primitives. The motivation of renaming the exchange functions is to ease system consistency checking, enhance readability and create different uses for the same exchange primitive. For example, "*x-opchan*" in the specification of microprogrammed architecture is used for different purposes: one for sending opcode and the other for receiving it. If we do not rename them in clauses 7 and 8 of Figure 3.11, when an error occurs to "*x-opchan*," it is perplexing for us to tell if it is an error of sending part or receiving part, and causes difficulties in debugging. Another advantage of renaming the exchange primitive is that we can add a mnemonic name and functional declaration of domain and range which clearly assists automatic system checking.

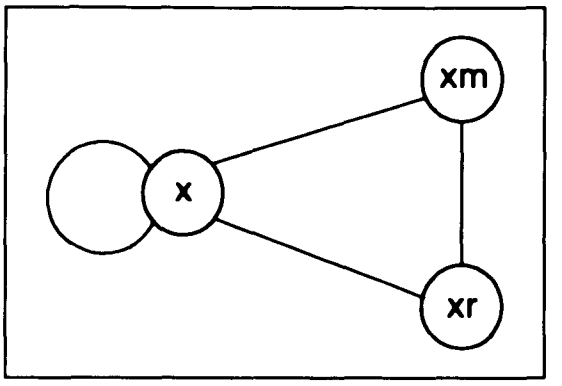

Figure 3.12 Match Relations of Exchange Types

Another application of the exchange primitives is to broadcast a message to all other recipient processes. To make sure that the message reaches every destination, the broadcaster can use

(j#1.. Num_of_Recipients < , x-broadcast-j[message] >) ;

to send the message, while each recipient process uses

xr-broadcast-j[message]

to receive message. Note that an xr-type function is used here to state that the recipient does not have to wait for the message. It is reasonable because broadcasting messages are usually sporadic events. This scheme actually can do more than just broadcast a single message because the recipient process can also feedback its own information to the broadcaster via message exchanging.

All previous examples are those of one-to-one correspondence synchronization, that is, only one

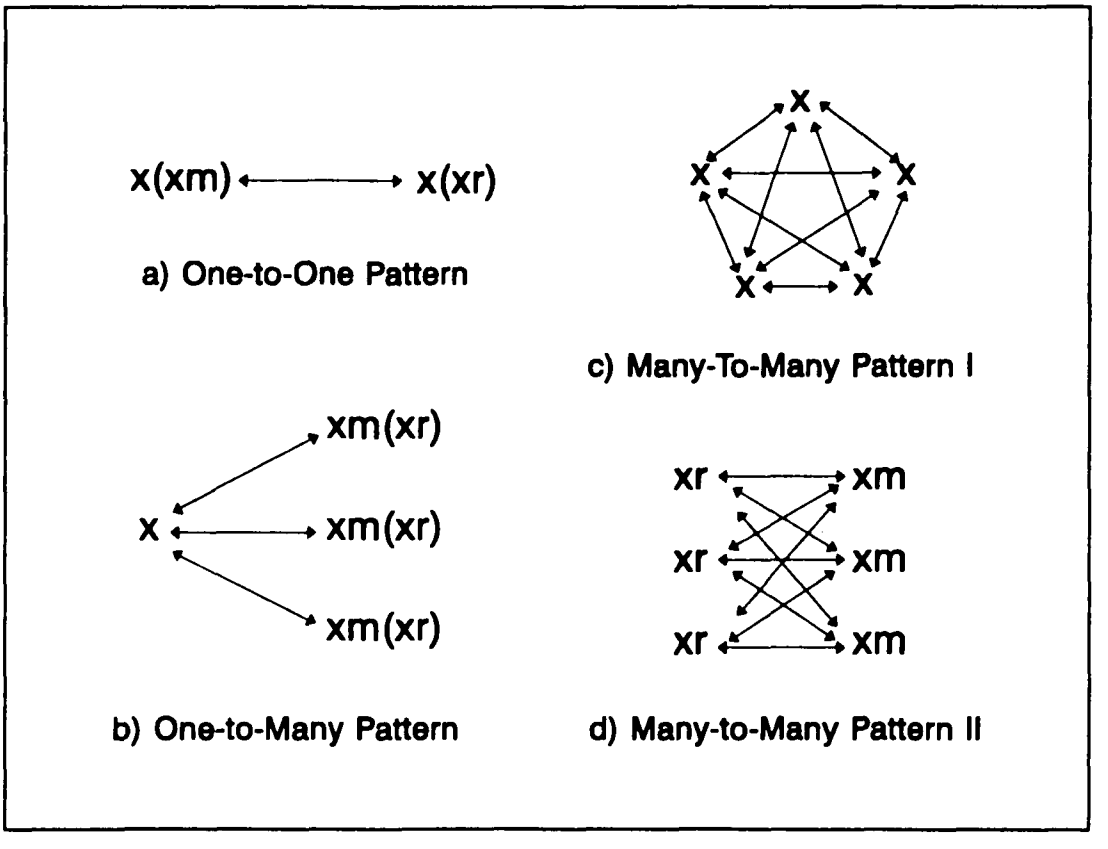

Figure 3.13 Synchronization Patterns of Exchange Functions

pair of channels exists for each possible match. Notwithstanding, the combination of three type exchange functions is capable of providing various, and more flexible, asynchronous communications. Figure 3.13 illustrates in principle how those combinations can deal with one-to-one, one-to-many and many-to-many synchronizing tasks. Of these diagrams, Graph c) shows a complete graph, indicating that, by using x-type channels, any pair of two processes is able to be synchronized because of x-type reflexivity, whereas Graph d) is a bipartite, exhibiting that an arbitrary xr-type channel can match any one of the xm-type's, but no two of the same xr-type or xm-type channels can match themselves because of non-reflexivity.

One well-known "one-to-many" example is the application of exchange functions as a synchronizing mechanism to an exclusively shared resource. Consider the specification in Figure 3.14. It specifies a well-known semaphore synchronizing mechanism employed for exclusive access to the single shared resource by multiple contenders. Clause 3 specifies five asynchronous and autonomous processes such as *shared-resource*, and *contender-process-j*, where j=1, 2, 3, 4. It describes such a situation: when contender j wants access to the shared resource, it first has to request an access right by testing the semaphore as shown in clause 8. Meanwhile, it has to set the semaphore to 1, meaning that the resource is no longer idle. It is similar to a test-and-set operation, realized by clauses 5 and 13, which exchange the value of the global bus lock with value 1. If the resource is free, i.e., the semaphore is zero, then it starts the utilization of the shared resource as specified in clause 10.1. Observe that the resource has to be released after the completion of utilization. That is another exchange operation of clauses 5 and 15 which clean the semaphore to zero. If the resource is already occupied by some other contender, semaphore will then equal one; contender j does not get a grant and has to wait and test again until the resource is freed. Hence, clause 10.2 simply returns the old contender's "*state*" to turn down the request and activates the next testing cycle. If two or more contenders simultaneously compete for this exclusively-shared resource, only one succeeds while the other gets "one" back for the testing because only one match will occur at any particular moment. Other xr-type "*swap*" exchange functions return their own argument without waiting. The PAISLey interpreter will determine the

```
#define  Num_Contenders  4                                                    "1"

SEMAPH-STATE = { 0, 1 } ;      " 0 - idle, 1 - busy "                         "2"

( shared-resource[0], j#1.. Num_Contenders < , contender-process-j[State] > ) ;   "3"

shared-resource: SEMAPH-STATE --> SEMAPH-STATE ;                              "4"
shared-resource[semaphore] = x-swap[semaphore] ;                              "5"

j#1.. Num-Contenders                                                          "6"
< ;
  contender-process-j: CONTENDER-STATE --> CONTENDER-STATE ;                  "7"
  contender-process-j[state] = process-j[(state, test-and-set-j[1])] ;        "8"

  process-j: CONTENDER-STATE*SEMAPH-STATE --> CONTENDER-STATE ;               "9"
  process-j[(state, semaphore)] =                                             "10"
            /equal[(semaphore, 0)] : release-j[resource-utilize-j[state]] ,   "10.1"
             True                  : state                                    "10.2"
            / ;

  release-j[state] = proj[(1, (state, clear-j))] ;                            "11"

  test-and-set-j: {1} --> SEMAPH-STATE ;                                      "12"
  test-and-set-j[x] = xr-swap[x] ;                                            "13"

  clear-j: --> SEMAPH-STATE ;                                                 "14"
  clear-j = xm-swap[0] ;                                                      "15"
> ;
```

Figure 3.14 Specification 7: A Synchronizing Mechanism - Semaphore

succeeder arbitrarily and fairly. The entire description exercises only fifteen clauses, concisely specifies in an intuitive and top-down refining method. This is among the most outstanding characteristics of the PAISLey language which implements a compact, executable specification and matches, faithfully, the initial intuitive design - a property which most current hardware description languages lack.

Referring to our system specification problem for mutual activities, we may consider the whole computer system as two entities in the simplest and most general way. Displayed in Figure 3.15, there are one processor, representing any unit that possesses computing ability such as CPU, I/O

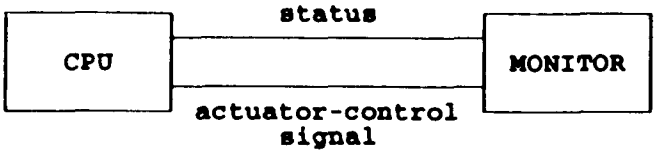

Figure 3.15 A Mutual Activity Model

Channel, and so forth, and a monitor that may represent a control unit for an external device, or a kernel of an operating system or other dedicated control program. Here, we concern ourselves only with their external mutual activities while ignoring the internal nature of those entities. The

```
"A template model for system mutual activities "

( processor[initial-processor-state], monitor[initial-monitor-state] ) ;

initial-processor-state: --> PROCESSOR-STATE ;
initial-monitor-state: --> MONITOR-STATE ;

processor: PROCESSOR-STATE --> PROCESSOR-STATE ;
processor[state] = simulate-processor[(state, get-actuator-signal[offer-status[state]])] ;

simulate-processor: PROCESSOR-STATE*CTRL-SIGNAL --> PROCESSOR-STATE ;

monitor: MONITOR-STATE --> MONITOR-STATE ;
monitor[state] = monitor-process[(monitor-state,
                                  send-actuator-signal[test-status[get-status[Null]]])] ;

monitor-process: MONITOR-STATE*CTRL-SIGNAL --> MONITOR-STATE ;
test-status: MONITOR-STATE*STATUS --> CTRL-SIGNAL ;

offer-status: STATUS --> STATUS ;
offer-status[status] = xr-status[status] ;

get-status: STATUS --> STATUS ;
get-status[state] = x-status[state] ;

get-actuator-signal: CTRL-SIGNAL --> CTRL-SIGNAL ;
get-actuator-signal: [signal] = xr-signal[signal] ;

send-actuator-signal: CTRL-SIGNAL --> CTRL-SIGNAL ;
send-actuator-signal[signal] = x-signal[signal] ;
```

Figure 3.16 Specification 8: A Template Model of System Mutual Activities

processor sends its state, which represents its working condition or monitors inquiring signals to the monitor. The monitor tests the state, determines its next operation, and then sends the actuator control signal back to the processor which is typically a request for interruption (or a data transferring request in a peripheral situation), a command execution or a system routine call in most control program environments. At some time point, the processor inquires these actual signals and decides whether to respond to their request or not, in terms of its own work condition and criteria. Figure 3.16 provides a PAISLey specification for such asynchronous activities. All functions are intuitive and self-explained. Just the functions "*processor-processing*", "*monitor-processing*" and "*test-processor-state*" may be defined elsewhere. Their details depend on the real situation and the target we want to design and specify.

Exchange functions are, by exception, a treasury of the PAISLey language that make the specification of concurrent multi-processing much more conveniently and naturally. It is usually a difficult task, when using most other computer hardware description languages, to specify an asynchronous processing, and especially difficult when specifying asynchronizing communication between several autonomous modules.

3.3. Performance Simulation by Execution of the PAISLey Specification

The term "operational" means that a specification is computerizedly executable. The aspiration of using an executable specification exists in order to conduct automatic system consistency checking, an early technique-independent system performance evaluation and design feasibility testing. In order to achieve such a goal, three factors - time constraints, running methods and scheduling strategies, and consistencies and performance measurements - take on an important role. The following section illuminates these factors separately.

3.3.1. Time Constraints in PAISLey

We have seen that PAISLey, by using applicative high order logic, has a strong power of expressiveness and descriptiveness in terms of computer systems. Nonetheless, for a long period of time, the traditional logic was mathematical tool confined to system verification and design analysis because of its weaknesses in timing specification. Computer systems are real-time systems because all digital systems are time-dependent. Thus, most performance requirements to be specified are time-related. In order to deliver a really measurable specification for the automatic system synthesis and coherence checking, we must have the ability to deal with these same time attributes. Many methods have been attempted to embed a workable timing specification into conventional logic. The method used in PAISLey represents one such attempt. PAISLey offers an explicitly specified time constraint statement that can be directly attached to a mapping function. The time attribute bound to a function refers to the evaluation time of the function attached. The evaluation time can be specified as a random variable with a constant, uniform, normal, exponential, or hyper-exponential distribution. Figure 3.17 displays typical time constraints that can be found in the PAISLey specification. Hyper-exponential distribution actually combines two exponential distributions of *mean-1* and *mean-2*, with *proportion-1*, a real number in the range (0, 1), indicating a proportion taken by the first exponential distribution with respect to the whole stochastic process. An upper or lower bound can also join a normal, exponential or hyper-exponential distribution to constitute a hybrid form.

Previously, we have seen the application of PAISLey's time constraints. One thing we want to point out here is that improper time constraints will cause system inconsistency. Looking back to Specification 3 of Figure 3.5 - a modulo-8 counter, clause 6 attaches a time constraint to function "*clock-cycle*." If we want to specify time limitation to function "*count-next-clock*," we should always comply with the law that it must be faster than that of "*clock-cycle*." Otherwise, the correct working behavior can not be guaranteed, and several clock cycles may be skipped without being counted because the speed of counting cannot meet that of "*clock-cycle*." Once the latter condition occurs, the PAISLey interpreter will detect this inconsistency and relinquish, producing a timing error indication, which actually depicts an internal system inconsistency

dilemma. Inconsistency of time constraints in PAISLey specification signifies that the specified objects are impossible to be constructed for the required performance.

In PAISLey, any mapping function occupies a certain amount of evaluation time to incorporate some degree of overhead in real world. The evaluation time of a mapping function with no time constraint attached is assumed to be arbitrary, but positive and bounded, unless it is set by the zero overhead or run in a bottom-up scheduling mode. Time constraints can also be attached to exchange functions. Though it can be connected directly, we have to annex time constraints to the renamed exchange function, if one is used. Then time attributes are committed to the renamed function. For example, if one wants to narrow the impulse length of clock cycle in the specification of a modulo-8 counter and place a time attribute onto exchange primitive, "*xm-impulse*," the following definition and time constraint may be used:

```
send-impulse[cp] = xr-impulse[cp] ;
send-impulse : ! lb 4.5 ns, ub 5.5 ns ;
```

Lower-bounds of time attributes to the real-flavor xr-type primitive mean that the exchange function waits at least the same amount of time and returns its own argument if no match occurs;

```
f:  ! lb 20.0 ns ;                                          "Lower-bound"
gate-delay:  ! ub 10.0 ns ;                                 "Upper-bound"
clock-cycle:  ! lb 20.0 ns, ub 20.0 ns ;                    "Constant"
seek-time:  ! uniform, lb 0.5 ms, ub 20 ms ;                "Uniform"
length-of-stay:  ! normal, mean 10.0 us, stdev 2.0 us ;     "Normal"
wait-for-serve:  ! exponential, mean 20.0 us ;              "Exponential"
packet-transfer:  ! hyperexponential, mean-1 10.0 us,
                      mean-2 20.0 us, proportion-1 0.77 ;   "Hyper-exponential"
inter-arrival:  ! exponential, mean 10.0 us, lb 2.0 us ;    "Hybrid"
```

Figure 3.17 Time Constraints in PAISLey

upper-bound to xr-type forces the exchange function to wait a certain quantity of time units, randomly between zero and the upper-bound, before returning when the match fails. Lower-bound time to x- and xm-type will make those exchange primitives take at least the number of the time unit even if the match might happen immediately after they are ready. Caution must be taken when an upper bound is tied to an x- or xm-type of change functions. It suggests that the match must occur and complete before this bound, otherwise a time inconsistent error will be printed to indicate that the time constraint is unable to be satisfied by all cases. It might be an internal system inconsistency or man-made over-constraint during specification.

Timing constraints in PAISLey effectively specify a variety of performance requirements directly. Since a process step usually corresponds to one cycle of the basic activity carried out by the process, and since external performance requirements are often associated with these basic activities, performance requirements are frequently expressed as timing constraints on successor mapping themselves such as we did in Specifications 1 and 2 for clock cycle. Figures 3.2 and 3.4 reveal the granularity of such a discrete simulation.

Another powerful feature of PAISley's time constraint is that it has an overall time scaling mechanism, which allows the designer to tune the relative evaluation rates globally among the sovereign processes. For example, in the processor and monitor situation as exemplified in Figure 3.16, we may want to test a different performance in terms of the relative speed between processor and monitor. The occurrence of monitoring events is usually much less than that of the processor's cycles which is why the interface is manufactured to cope with this disparity. Accordingly, we can use command "*setscale 10 2*" to alter the relative rates of processes "*processor*" and "*monitor.*" The command settles the evaluating rate of process 2 - "*monitor*" ten times more slowly than before by resetting time scale factor of process 2 to 10. The default time scale factor for every process is 1. It is very convenient to test the overall system performance in terms of different time scale factors even when there is no need to add or alter the time constraints in specification.

3.3.2. Running Methods and Scheduling Strategies

The PAISLey interpreter is a common UNIX™ application. Hence, if Specification 1 of Figure 3.1 is placed in a file "clock.pai," then it can be run by typing regular UNIX™ command "*paisley clock.pai*." If no error or inconsistency is detected, the execution can be started immediately and the trace, something like Figure 3.1, will be obtained. More generally, if a PAISLey specification is placed in multiple files: file-1, file-2, ..., fine-n, then it can be executed by one of the following commands:

paisley file-1 file-2 ... file-n (foreground)

or

paisley -s script-file file-1 file-2 ... file-n >& trace-file & (background)

The PAISLey language is engaged in a conversational environment. When specification is run in a foreground method, commands such as "*start*," "*continue*," "*quit*," etc., can be entered interactively. The trace of execution is printed to UNIX™ standard output device "stdout," whose default is usually a video screen. PAISLey provides a command "*settrace on file-name*" to redirect the tracing output to a user defined file "*file-name*." However, if the PAISLey specification is executed in a background method, all commands to be used must be placed in a script file. In this ground, the tracing results can be redirected by a regular UNIX™ redirection mechanism such as "*>& trace-file*," used in the above example. Observe: two background signals are operated here. Parameter "*-s*" indicates the background mode to the PAISLey interpreter while the symbol "*&*" at the end of the command is the convention of UNIX™ background jobs.

We have known that the PAISLey specification is composed of asynchronous processes which are

specified in a system declaration statement and operate concurrently during execution. Each process is an autonomously active computation entity and carries out succeeding mapping. That is, each process is specified on a state space and starts its evaluation at an initial state in it. The resulting value, called a successor state, will be taken as a new argument and the same evaluating process will be applied to it to yield the next successor state. This process will be repeated cyclicly ad infinitum, unless some time or space limitation, set by user or boundness of UNIX™ system resources, is reached. Figure 3.18 displays such a successor mapping. The interval between two consecutive successor states is called a process step. The timing length of each process step is determined by the scheduling strategy adopted and the time constraints affixed. These self-governing processes are able to communicate with each other or synchronize by employing exchange primitive functions.

There exist two scheduling modes in PAISLey which can activate each successor mapping step. The default is a *top-down* mode in which each process's step length is calculated at the beginning of the step. It is the product of three factors - the basic time unit which is set by "*setbtu time-unit*"; the process's scale factor; and a pseudo-random number between zero and one (assuming that no specific time constraint is attached to this process). By the formula, we see that basic time units and scale factors jointly influence the rate of successor mapping. A larger time unit or a larger scale factor lets a process run slower. Normally, the time unit is determined by the timing property of the object to be designed - say nanosecond to digital circuits and millisecond to peripherals and so forth - while the scale factor is decided by the relative rates between processes as we have seen in the previous section. The explicit time constraints attached to the process will override this formula because the specified timing bound must be satisfied.

The other important scheduling strategy is a *bottom-up* mode in which every event in the process is scheduled to be executed as soon as it is logically enabled, contingent only to time constraints and minimum overheads whose default is zero and adjusted by the "*setoverhead*" command. The *bottom-up* strategy truly simulates the events which occur in a sequence of discrete time points

and especially fits the digital systems, whereas the *top-down* method is good at controlling the average relative rates of the specified processes. We can use the following command to choose the scheduling strategy

setscheduling bottomup | topdown <list-of-process-numbers>

To erect, for instance, the bottom-up strategy to processes "*clock-cycle*" and "*counter*" in Specification 8 (Figure 3.5), we can use command "*setscheduling bottomup 1 2*" before starting the execution of it, for a discrete-event simulation.

Earlier in Section 2, we mentioned that PAISLey tolerates incompleteness in the sense of being

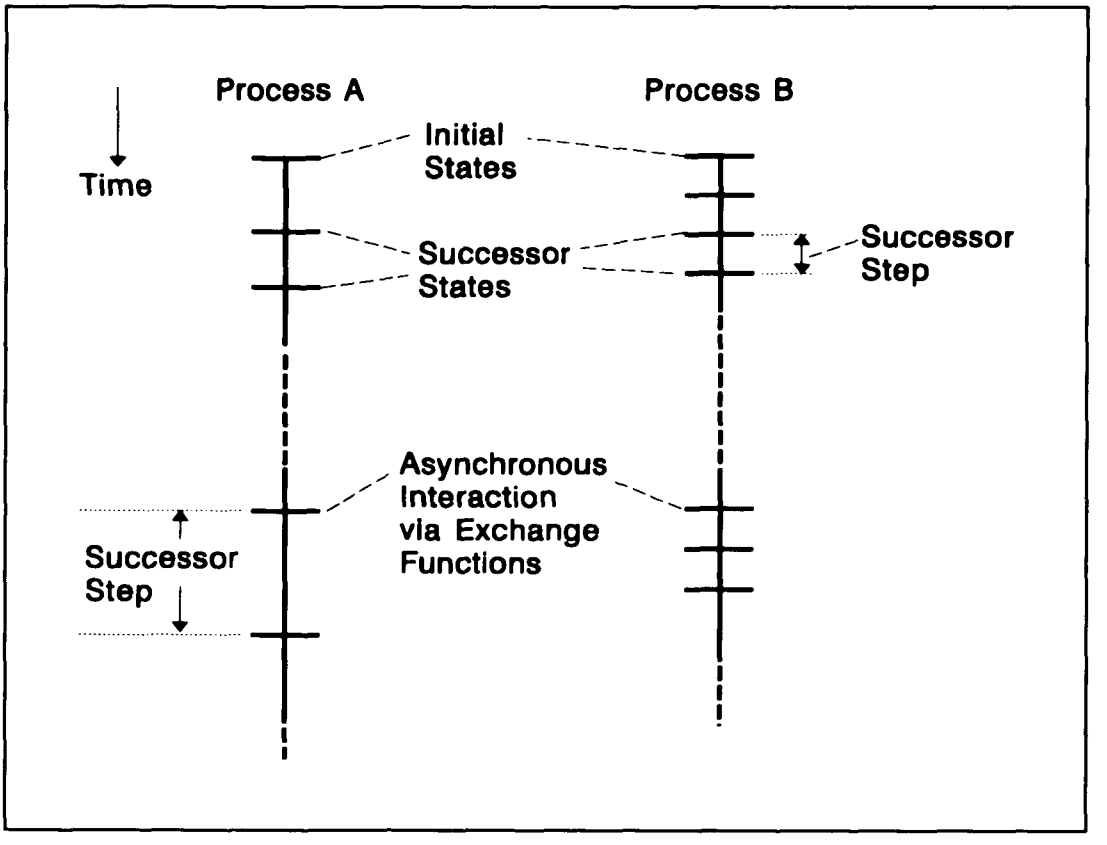

Figure 3.18 Processes in Action

able to miss some set definitions, function declarations and definitions. Furthermore, with PAISLey, another mechanism is furnished to assist in the respect of the incomplete specification. The PAISLey interpreter adds an ability to inhibit the evaluation of individual mapping process with no need to modify their specification. This can be done by the "*setprocess off <list-of-process-number>*" command. All the functions defined in the off processes are deactivated. PAISLey treats the deactivated function as an undefined one and, instead of evaluating, it picks up a value randomly or constantly from the range of the function, if it is declared, when that function is encountered during execution. The communication channels also can be turned off by the "*setchannel off <channel-name>*." When channel evaluation is needed, the "*ANY*" value is returned if the channel is not renamed, otherwise one value chosen from the range of the renamed function is returned. It is in this sense that we encourage users to make function declarations for domain and range and rename exchange channels, despite the fact that both can be ignored. With the renamed channel, and defined domains and ranges, the PAISLey interpreter is "intelligent" enough to pick up the right values even when those channels and functions are deactivated and can handle the missed mapping processes integratedly. It is a valuable facility to debug a specific flaw or to test the exclusive effects of certain specified objects because it permits easy isolation of several functions during the execution.

Before leaving this section, it should be indicated that the files in the "*paisley*" execution command are unnecessary to be put solely in PAISLey specification. It could easily be a compiled C executable module. The C function processes and the PAISLey processes run concurrently. The C code must include "*pdefines.h*," as described in Section 3.2.1. During the compilation, the directions of library functions and header files of the PAISLey and C interface also must be designated in the compiling parameters.

3.3.3. Consistencies and Performance Measurements

The primary purpose of running a specification under computerized environments is to espy all

possible inconsistencies and assess early performance measurements aggregated through execution before real implementation. Recognizing those inconsistencies is significant because it represents the specified system is constructible in terms of performance requirements. In fact, like most current automatic systems, the PAISLey interpreter performs a static analysis and detect all inconsistencies found via static analysis, including all syntactic inconsistencies, referring to syntactic and grammatical errors; all value inconsistencies referring to the conflicts among definitions, declarations and applications of mappings; all process-level inconsistencies, referring to those in the attributes of a process, typically, timing contradictions within the process; part of system-level inconsistencies referring to those occurring between processes. The most frequently detectable system-level inconsistencies are those caused by improper use of exchange functions. The majority of system-level inconsistencies, such as deadlock - caused by exchange functions and global timing inconsistencies among processes - are only locatable via dynamic analysis. In other words, they can be identified via the analysis of the tracing results. It is not a difficult task to find these pitfalls in the PAISLey environment, because the trace can list the initiation and termination of all processes, mapping functions along with their arguments. The PAISLey interpreter also affords the command "*setreport*" to let the designer staff trace individual successor mapping and collect useful information for debugging and measuring.

However, consistency checking merely resolves one aspect of the system design; that is, whether the specified system is constructible. It does not answer whether the specified system is satisfactory and profitable, or whether it works correctly and reaches the design goals; ideas which should depend upon measurement collection and performance evaluation. A variety of information can be collected from the trace of execution. We have to choose those which mostly reflect the designers targeted objects. The real method varies on realistic worlds, specifying methods and personal styles. Here, we must turn to several previous specification examples; more concrete and comprehensive instances will be seen in the following section.

Figure 3.19 captures the starting section of the trace from the execution of Specification 3, a

modulo-8 counter. In this case, we want to know if the counter catches up every clock cycle and if the state of the counter is progressed sequentially and circularly. It is clear to see that the specification satisfies those performance requirements. The clock cycle is generated every 2 time units, that is, 20 ns (2 x 10 ns), and the specification is running in a bottom-up scheduling mode (namely, the evaluation of every function is scheduled as soon as possible - a standard discrete event simulation). Many factors such as overhead, scale factor, time constraint, and scheduling strategy will affect the correctness and performance. If we desire, for instance, to test the counter in a circumstance involving randomly occurring clock signals, we need to alter the time attribute affixed to "*clock-cycle*". Instead of a constant value, we may specify an exponential distribution with a lower bound, say "*clock-cycle: ! exponential, mean 10.0 ns, lb 5.0 ns;*." Due to some technique restriction, the counter consumes approximately 10 ns of delaying or propagating time. The time constraint has to be appended to process "*counter*," or to the functions in it. Let us add clause "*count-next-clock: ! uniform, lb 9.5 ns, ub 10.5 ns;*" into Specification 3 and execute it. We now obtain the trace given in Figure 3.20, which clearly shows that several clock cycles are lost because of the counter's slower operation.

```
> start
Time          1          2
in 10 ns

   0.00000    clock-cycle[1]
   0.00000               counter=1
   2.00000    clock-cycle[1]
   2.00000               counter=2
   4.00000    clock-cycle[1]
   4.00000               counter=3
   6.00000    clock-cycle[1]
   6.00000               counter=4
   8.00000    clock-cycle[1]
   8.00000               counter=5
  10.00000    clock-cycle[1]
  10.00000               counter=6
  12.00000    clock-cycle[1]
  12.00000               counter=7
  14.00000    clock-cycle[1]
  14.00000               counter=0
  16.00000    clock-cycle[1]
  16.00000               counter=1
  18.00000    clock-cycle[1]
  18.00000               counter=2
  20.00000    clock-cycle[1]
  20.00000               counter=3
    ... ...          ... ...
```

Figure 3.19 Trace of the Execution of the Modulo-8 Counter Specification

In a more complex specification the analysis of tracing results is not always so straightforward. Once an inconsistency or a malfunction is spotted, scrutiny must be borne. One should think over

what is the real reason causing such a problem. It could be an internal error, referring to design deficiency; or a specification error, referring to over- and under- constraining the target object during specification. Multiple revising passes may be required for either design modification and improvement if there exists an internal system inconsistency, or when specification correction is needed should there exist bugs.

```
> start
Time          1          2
in 10 ns

 ... ...      ... ...
 12.19926     clock-cycle[1]
 13.21552                counter=0
 14.26128     clock-cycle[1]
 14.72309                counter=1
 14.76128     clock-cycle[1]
 15.72145                counter=2
 16.39265     clock-cycle[1]
 17.40534                counter=3
 17.57158     clock-cycle[1]
 18.55923     clock-cycle[1]
 18.59437                counter=4
 19.89005     clock-cycle[1]
 20.93375                counter=5
 23.22501     clock-cycle[1]
 23.72501     clock-cycle[1]
 24.19361                counter=6
 24.90887     clock-cycle[1]
 25.88920                counter=7
 27.00602     clock-cycle[1]
 27.43954                counter=0
 27.58242     clock-cycle[1]
 28.63021                counter=1
 ... ...      ... ...
```

Figure 3.20 Trace of the Execution that Detects Performance Deficiency

As another example of testing average performance - a partial trace from the execution of Specification 7 for a synchronizing mechanism - semaphore - is exhibited in Figure 3.21. To be clear, only the termination of the "*test-and-set-j*" functions of contender 1 and 4 are reported, which symbolize the acknowledgement of shared resources since these are renamed exchange functions and return the current value of semaphore at the end of exchanging. Recall that 0 indicates the idle status and 1 means the busy status of the shared resource. From Figure 3.21, we can discern that "*contender-1*" issues sixteen requests and obtains eight access permissions by receiving a semaphore value 0, while "*contender-4*" has five out of nine. The success ratio is approximately 50% and 50%, respectively. There are more tests coming from "*contender-1*" from "*contender-4*." It is because we attach a faster time attribute to the former, with an exponential distribution of mean 12.0 μs, versus an exponential distribution of 8.0 μs to the latter. The average waiting time due to busy shared resources is approximately 14

```
> start
Time          1        2        3        4        5
in 10 us

     0.16480          test-and-set-1=0
     0.66226          test-and-set-1=0
     1.53359                                    test-and-set-4=1
     1.93759          test-and-set-1=0
     1.95334          test-and-set-1=1
     2.91493          test-and-set-1=0
     4.17851                                    test-and-set-4=1
     4.93652          test-and-set-1=1
     5.07685                                    test-and-set-4=0
     6.45428          test-and-set-1=1
     6.89692                                    test-and-set-4=0
     8.84017                                    test-and-set-4=1
     9.00336          test-and-set-1=0
     9.42895          test-and-set-1=1
     9.63755          test-and-set-1=1
     9.79570          test-and-set-1=1
    10.24674          test-and-set-1=0
    10.60178                                    test-and-set-4=1
    10.85564          test-and-set-1=1
    12.35030          test-and-set-1=0
    12.83504          test-and-set-1=0
    12.90498                                    test-and-set-4=1
    12.95563                                    test-and-set-4=0
    14.09056          test-and-set-1=1
    14.95649                                    test-and-set-4=0
    15.39648                                    test-and-set-4=0
     ... ...           ... ...                  ... ...
```

Figure 3.21 Trace of the Execution of Specification 7 - A Synchronizing Mechanism

microseconds within a 154.0 microsecond elapsed time[1]. This is computed by the equation

$$Average\ Waiting\ Time = \frac{1}{N}\sum_{i=1}^{N}(T_{i_0}-T_{i_1})$$

where N is the number of successful requests; T_{i_0} is the time at which the i-th "*test-and-set*" command succeeds and T_{i_1} is the time of the first failing test command since the last, that is, the

[1] We set the time unit as ten microsecond (10 μs) during the execution. Subsequently, the elapsed time is approximately equal to (15.40-0.00) x 10 μs = 154.0 μs.

(i-1)-th, successful one or is the same as T_{t_i} if there is no failing test. Analogous to this, the average waiting time for "*contender-4*" is about 25 microseconds. The summary information can be easily obtained via UNIX™ shell commands such as cat, grep, ..., or any of the other computing utilities.

In spite that performance measurements rely heavily upon the outside world to be designed and specified, collection of these measurements from execution of the operational specifications actually counts on the design staff's objectives, experience and personal specifying style. The unique way of gaining that is application and practice.

4. THE WLAB CACHE SYSTEM

People have spent decades trying to improve memory performance and attempting to shrink the speed gap between computing processors and memory devices. Yet, even the advent of cache and semiconductor main memory is unable to make the gap significantly smaller due to the rapid speed of modern processors and the associated parallel processing. Presently, the multi-processor systems are getting even more popular because of the swiftly dropping costs and the quickly increasing power of processor. Dual processor workstations, mainframes, even quadruple processor ones are no longer rare. This situation makes the use of a high quality hierarchical memory system even more integral because most contemporary multi-processor systems are tightly coupled by using a single shared main memory and are required to deal with the job at more than twice the speed as a single processor because two or more processors are accessing the same memory. Two big problems seriously obstructing the high performance of the cache memory system are: the miss latency problem and the cache coherence problem. Both of these boost memory bus traffic and slow down the global speed of computer. These two problems become even more aggravated in the case of tightly-coupled multiprocessor systems, since several processors might simultaneously compete for the exclusively shared memory. Reading a missing block and writing to shared blocks will both affect and block other processors in the systems.

A small fast set-associative memory, which functions as a write multi-buffer between the local caches and the shared main memory in a tightly coupled multi-processor system, is proposed in this section to reduce the write through and miss penalties, and to increase the memory bus bandwidth. It has already been recognized that a single buffer is not sufficient to cushion the data to be written back (write-back) or through (write-through) produced by write operations. Write multi-buffers seem mandatory now, particularly in multiprocessor system architecture which

employs a shared memory structure. However, computer architects are still reluctant to use the write multi-buffer for it normally complicates the cache coherence problem. Borrowed from the concept of the virtual storage, the cache hierarchy [PrzH89, WanB89] is developed and is expected to tap the latent power of caches. Nevertheless, by adding one more level of cache, two or more levels of fast address mapping mechanisms are required, which introduce high complexity and expenditure.

The cache organization presented in this dissertation is used for a shared memory multiprocessor system using one global fast and small set-associative storage device. Although a small fast associative memory, it acts like a multi-buffer rather than a second level cache. We prefer to call it a Write Look-Ahead Buffer, or a WLAB for short, since it is neither a conventional cache nor a so-called cache hierarchy. The motivation here is that, by applying a single global fast associative memory to function as a typical write multi-buffer, we can reduce the write through penalty and the miss latency, and also easily solve the cache coherence problem, whereas a regular write multi-buffer introduces much more complexity to the resolution of the cache coherence for the shared cache blocks, while normally prevents people from using it in the cache coherence situation. Fortunately, we discover that the associative searching property of the new Write Look-Ahead Buffer is able to resolve the cache coherence implicitly and effectively. As a bonus, the introduction of the Write Look-Ahead Buffer will supplementally decrease read miss latency by directly reading those blocks from the fast speed Write Look-Ahead Buffer instead of the slow shared memory. In addition, a hardwired local bus semaphore is added to each local cache controller and a global arbitrator mechanism is provided to choose the competitors, greatly facilitating the bus cooperation and reducing the bus traffic caused by bus synchronization contention.

Based on this cache organization, a snooping-based cache protocol using write-through with or without broadcast strategies is introduced. Yet, it needs not be this strategy. The write-back or write-through with other combinations works as well. We choose this one because it resolves data

content inconsistency smoothly and conveniently without major performance impediments and without the complexity posed by most snooping-based protocols. Additionally, the write-through technique is going to be dominant in multi-processor systems and is also expected to raise hit ratio and reduce miss latency. The protocol can be implemented easily compared to other cache coherence protocols [AgaS88, ArcB86, Fra84, Goo83, ParP84, Pre91]. The simulation and the performance evaluation has been carried out with the help of the PAISley specification [HabJ90, 92a, JinH92, Zav82a, 91].

Both Sections 4 and 5 are dedicated to the Write Look-Ahead Buffer hierarchical cache system. They are organized as follows: Section 4.1 shows the organization of the WLAB structure; Section 4.2 discusses the possible configurations of the Write Look-Ahead Buffer in regards of the local caches and presents two generic solutions; Section 4.3 summarizes the cache coherence protocols based on the organization illustrated in the previous two sections; Section 4.4 considers the cost and performance and analyzes the expected results from the WLAB cache system.

Section 5.1 demonstrates how PAISLey saves the specification for the Write Look-Ahead Buffer cache system - a real world example. Section 5.2 presents simulation results obtained from running the PAISLey specification. The last two sections, together, shed light on how performance evaluation can be carried out and valuable measurements can be acquired immediately from executable specifications.

4.1. The WLAB Cache Memory Organization

Figure 1 draws a diagram of a Write Look-Ahead Buffer cache memory architecture in a four processor system. A small fast multi-way set-associative memory is placed between a single shared memory and local processors. We name it a Write Look-Ahead Buffer, or merely, a WLAB, because it actually acts like a write buffer. All the blocks or words to be written through

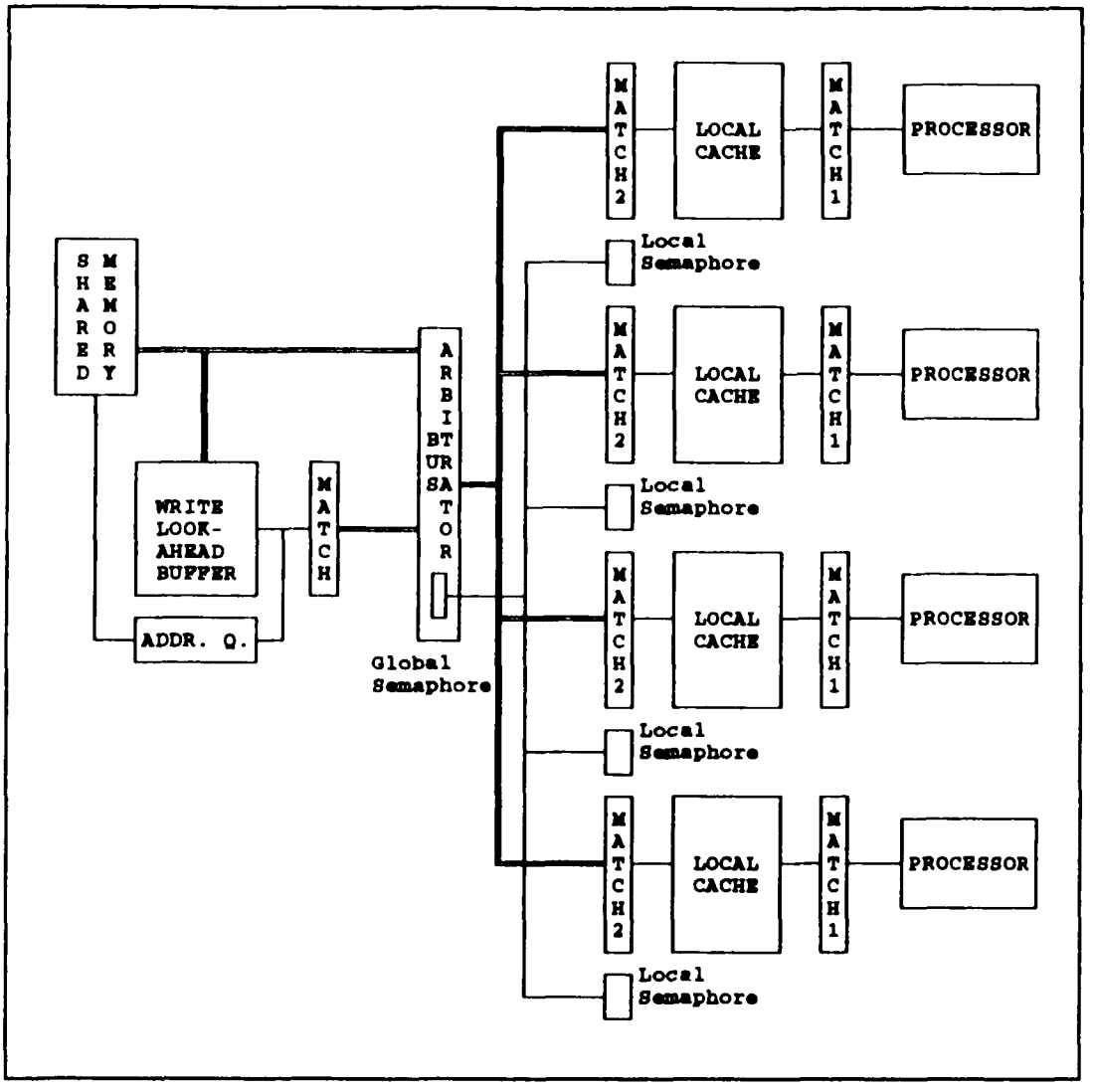

Figure 4.1 The Organization of a Write Look-Ahead Buffer Cache System and Hardwired Semaphores

to memory will be stored in this buffer where they wait for their turn. The memory control device looks up the address on the queue along with the buffer and moves the data from buffer to the memory. All read misses consult the buffer too. The block is read from the buffer instead of memory if the missing block is in it. Therefore, the latency of miss operations is reduced since the buffer is a much faster associative memory. All the shared blocks in other private caches can be either updated via broadcast or set to "invalid," if no broadcast, when a write operation occurs. However, if the access to those blocks is made shortly after they are disabled, the valid block,

with most possibility, is still in the write buffer, and thus is read quickly from it. It speeds up the write through operation, reduces the miss latency and subsequently increases the shared memory bus bandwidth, while at the same time resolving the cache coherence problem appropriately and effectively.

The Write Look-Ahead Buffer basically is a multi-way set associative cache, working functionally in a different manner from a regular cache. It will not, for instance, read the block into the buffer when a miss read takes place. The possible generic configurations of the buffer, with respect to the structure of the local caches, are presented in the next section. The write through strategy, adopted in cache protocol during our simulation, is applied to all the private caches of local processors. For each write operation, no matter whether it is a write miss or write hit, the data are written to the Write Look-Ahead Buffer using set-associativity rules. If it is the only unwritten back block in the buffer, it will be written back to the shared memory immediately, via the internal bus between the shared memory and the look-ahead buffer. If the Write Look-Ahead Buffer is not empty, the contents of the block are stored in the buffer and wait its turn. The writing address is placed in the address queue, which is built on a First-In-First-Out structure. If the writing is a duplicate one, i.e., the block is to be written through due to the previous operation, but has not been done so far, the writing address is inhibited from the queue as it already exists and only the block of look-ahead buffer is refreshed. The detection of the duplication is convenient because of the look-ahead features. The matching signal will tell it and can be used as the address prohibiting signal.

Each write operation in this architecture will write through the data to the rapid Write Look-Ahead Buffer instead of to the slow shared memory and let the physical memory writing be executed in parallel with the local processing during the memory's idle cycles, so the write penalty is decreased due to fast buffer speed. Despite a regular write multi-buffer which may reduce the write through penalty and the write miss latency, it greatly complicates the cache coherence protocols since the write buffer will include the modified blocks that might be requested by a read

miss from the other processors in subsequent memory accesses. In addition, many redundant blocks with as few as one byte may exhibit differences as a result of consecutive write operations, which clearly throws in more traffic for the memory bus and lowers the memory performance, especially during the write burst. Thanks to the property of associativity, the Write Look-Ahead Buffer solves this problem conveniently and favorably. First of all, the dirty blocks can be read from the Write Look-Ahead Buffer directly by virtue of the associative mapping, which eliminates the complicated searching strategies and circuits required by regular one. Secondly, as we mentioned in the previous paragraph, if the write through block is a duplicate in the buffer, no new block will be put in the buffer, only the dirty bytes will be replaced. The same principle applies to the word write through technique. Hence, it eases the bus traffic and relieves the burden of the memory device borne heavily during the write burst.

For read miss processing, the missing block will be looked up simultaneously in the Write Look-Ahead Buffer, other caches and memory. If the block is not in the buffer and caches, the block is loaded from the shared memory, functionally similar to most cache systems. If the block is in the buffer, it is the most-recent copy and is loaded directly from the buffer. Therefore, the Write Look-Ahead Buffer will never add any data inconsistency complexity, as do many write multi-buffers. Besides, reading the missing blocks directly from the look-ahead buffer shortens the read miss latency. This is a significant improvement and a favorable operation since the look-ahead buffer possesses a more rapid speed than that of main memory. Searching the look-ahead buffer will not cause any extra overheads since the memory and buffer accesses are executed in parallel, also the associative characteristic of the buffer quickly determines the presence of a block in the buffer with no need for complex coherence approaches.

Two matching circuits work independently and simultaneously in each local cache of a processor. One of them, matching circuit 1 in Figure 4.1, is the normal one that performs the matching of the addresses, produced by the local processor, to the blocks in the local cache. The other one, matching circuit 2, performs the matching of the identifier of the block on the bus to the blocks

of the local cache during a write through operation. If a match occurs, that means one of the local cache blocks is shared with the one to be written through. The matched local block will be either set to invalid or refreshed by switching the data on the bus to its local cache. We leave this choice to the strategy determination of a cache coherence protocol. The latter case causes one cache cycle delay for a match. However, when two or more matches occur on several private caches at the same time, the switching data to those caches is executed simultaneously with no cascade delay affect, since the data are broadcasting to those local caches. If no match occurs in a local cache, the local processing is performed continuously without any delay. The shared block matching processing is transparent to local processors because the second set of matching circuit belongs to the part of a snooping cache controller that takes care of bus traffic listening and testing, and operates autonomously.

A local hardwired semaphore is introduced to the bus interface of each local processor and a global hardwired one is placed in the bus arbitrator. Each time the contents of the global semaphore are changed, the value will be broadcast to every local semaphore at the same time except the one that is the source making the change occur. The bus arbitration is simple and similar to the synchronizing mechanism which we have discussed and specified in Section 3.2.3. The local processors with their local semaphores equal to zero are able to compete for the bus by issuing an exchanging signal 1 which is placed in the contents of the global semaphore, as a kind of the test-and-set processing. Two or more processors can test the value of the global semaphore at the same time, but only one of them will receive 0 response in an arbitrary, but fair, mode, while all the others will get 1 response and have their local semaphore set to 1. The succeeding processor which gets 0 response acquires the bus control, starts its bus activity and sets the global semaphore and the other local ones to 1 in the meantime. When it completes its bus transaction, it resets the global semaphore to zero and broadcasts this value to all other local semaphores. Thus, another cycle of the bus competition can start since then. Combining the contents of the global semaphore and the local ones as the states of each local processor, we may get a finite state machine, as depicted in Figure 4.2, for each local bus synchronous processing.

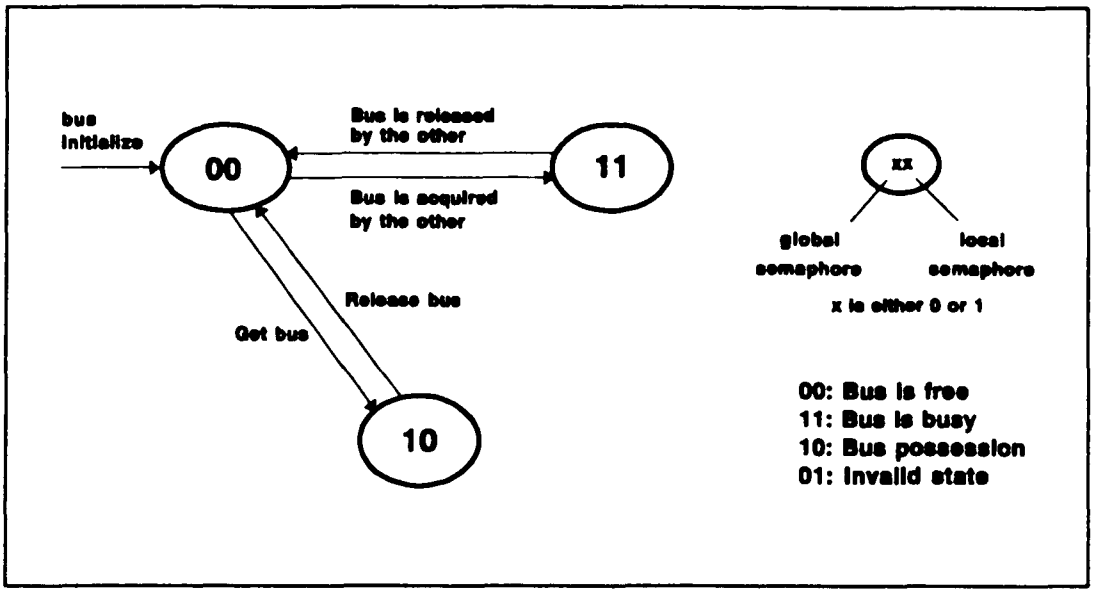

Figure 4.2 Finite Automata for the Status of Semaphores

The advantage of using the local semaphores is that a processor does not have to keep testing the global semaphore through the bus while one of the other processors is performing the bus transaction, unless the local semaphore is again cleared to zero. This reduces the bus testing traffic and increases processing speed. The introduction of the hardwired local semaphores also makes the synchronization processing transparent to those programmers who are not interested or do not have the responsibility for bus coordination, and allow them to concentrate more on achieving a higher degree of parallelism.

4.2. Configurations of the Write Look-Ahead Buffer

The size of a Write Look-Ahead Buffer may be very small, depending on the speed dispatching ratio of main memory to caches and the number of processors in use. By virtue of the inexpensive cost of hardware chips for relatively small associative registers, a Write Look-Ahead Buffer can be made a little bit larger than necessary to get high performance with no significant

price increase. The interesting question here is what should be the configuration of the Write Look-Ahead Buffer, with respect to those of the private caches in local processors, and what relationship of the global WLAB and local caches should keep. Many different configurations can be found in terms of the set associativity, size and the relationship regarding the structure of local processor caches. Two generic Write Look-Ahead Buffer configurations are proposed in this thesis; their relations with the local caches are sketched in Figure 4.3.

For simplicity, we assume that all the private caches in local processors have the same features for uniformity, though this is not mandatory. Let the local cache be a K-way, set-associative memory and let S be the number of sets. Suppose there are N parallel processors sharing a single global memory. The first WLAB configuration adopts a c·N·K-way set-associative memory with the same number of sets as that of the local caches. Here, c is a positive fractional or an small integer, usually less than 2. Truncation, or round up, will be done if c·N·K is not an integer.

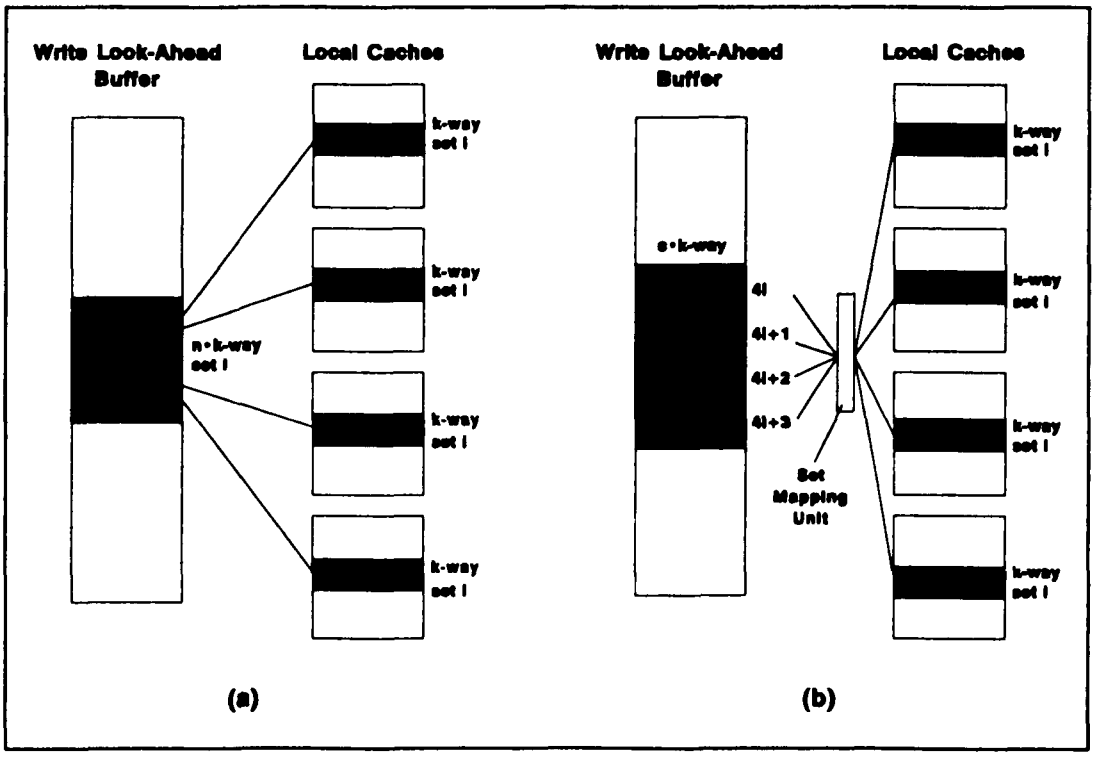

Figure 4.3 Configurations of the WLAB with Respect to Local Caches

An advantage of this configuration is that the blocks from the local caches will go to the same set of the Write Look-Ahead Buffer, no set or address translation is required. Diagram (a) of Figure 4.3 illustrates this relationship between the Write Look-Ahead Buffer and the local caches by letting c equal 1. The shaded rectangles indicate sets with the same number. In a worst case, it deals with c·N·K blocks from the same set of the local caches without writing even one back to the memory. On average, each private cache can send c·K blocks alternatively with those of the other caches from the same set - an almost impossible event in the real situation for a proper value of c. As a result of locality, the address space generated by a write burst focuses mostly on very few blocks. One may worry that two multiplications would make the product of c·N·K too big to get a cheap, high performance cache. According to Stone [Sto87], N, the number of processors, cannot become too large because of overhead. The real systems rarely exceed 16. Besides, most popular modern caches are 2 to 16 way set-associative, the product will yield a 4 to 256 way set-associative cache, which is an acceptable solution. Another merit of this organization is that adding new processors to the system will not affect the look-ahead buffer's capacity of dealing with the write buffer collision, though the original relation may be violated. Of course, the chance of the write block collision, seen in the later explanation, may be increased if too many processors are added.

The second configuration, shown in diagram (b) of Figure 3, uses a c·K-way set-associative memory with r·N·S number of the sets, for small positive numbers c and r. The block in the i-th set of a local cache can be stored in one of the r·N write buffer sets, say 4·i, 4·i+1, 4·i+2, and 4·i+3 in diagram (b) of Figure 4.3, where N=4 and r=1. A set mapping unit is needed here to transform the set number and tag between the global buffer and the local caches. It is really a simple translation, just an easy decomposition. By the same example in diagram (b), the first two bits of the tag in the local cache could be used to determine the set of the WLAB in which the physical local block data should be placed (see Figure 4.4). The tag of the local block could be stored in the global buffer without any change in order to simplify the set mapping unit. However, the example illustrated in Figure 4.4 stores the tag to the global WLAB by knocking

out the first two bits of the local tag to save the space and cost of the Write Look-Ahead Buffer. We do not concern ourselves about the concatenation of these missing bits of the tag when the block is read from the global buffer to the local one, since this is rendered unnecessary because the complete tag or address is always on the bus during the time the access miss occurs from the local caches.

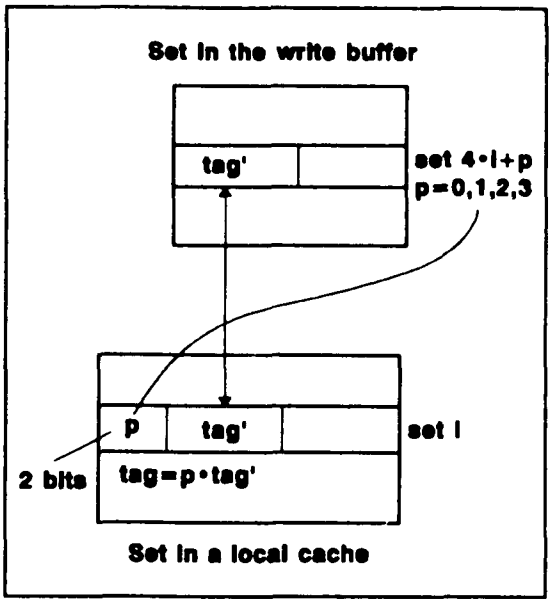

Figure 4.4 Set and Tag Mapping in Diagram b of Figure 4.3

The associativity is now reduced because of the c·K-way set instead of the c·K·N-way set in the first configuration. If a large associativity is used for the private caches in local processors and a high degree of parallelism, that is, a large N, is committed in the multiprocessor system, this configuration might be a more economical and feasible solution for a high performance Write Look-Ahead Buffer. However, one problem exists in that the block from the local cache probably will be unable to be placed in the same set of the global Write Look-Ahead Buffer. A simple set number translation is required between the local cache and the WLAB, as discussed before. This unit could potentially extend the clock time of the look-ahead buffer access cycle. In addition, the capacity of the second configuration to deal with the worst cases of write buffer collision is not the same as the first solution. The Write Look-Ahead Buffer collision occurs when all the blocks in a set of the WLAB are occupied, while the new arrival of the write through block, which is not in the WLAB buffer, is mapped to this block, too. Though this case should be rare indeed, much more infrequent than the write miss of a regular cache in a well-organized Write Look-Ahead Buffer, a resolution of the collision and a discussion of the possibility is given in section 5. As the blocks with contiguous tags in the local caches will be switched into the different sets of the global buffer, the possibility of this collision is even lower

than that of the first configuration. Hence, the overall performance of the two configurations in regard to this aspect should be close.

As a convenient illustration, we use integer for both c and r. In fact, one may use any real number for this purpose as long as the product is rounded, truncated or ceiled to the nearest integer, or to a power of 2 if necessary. For a smaller N, a memory that is identical to the local cache would likely be used as a global Write Look-Ahead Buffer, an extreme case of the previous generic configuration by letting c·N equal 1, which possibly makes c a fraction.

As we mentioned at the beginning of this section, as a write multi-buffer, the size of a WLAB could be relatively small in contrast with that of local caches. By assigning a very small fraction to c in the first configuration, the number of sets of the WLAB can be made the same as those in local cache with fewer lines in each set. It saves the cache address conversion and simplifies mapping circuits. When the number of sets in a local cache is too large to keep a small WLAB, the second configuration can be adopted and the number of sets in a WLAB can be diminished by a 2^k factor which is a reciprocal of r·N, for some k>0. Conversion of addresses between the WLAB and caches is now required, but can be done by a simple decomposition or concatenation. Certainly, this conversion needs the support of a cache matching circuit.

Another interesting case is when N is equal to 1, which is really a single processor system. With the help of the Write Look-Ahead Buffer, we believe that the hit ratio will be improved and the miss latency will be reduced for this case as well. The simulation results confirm our expectation. Particularly, it gains significant benefits in a cold-starting machine or in a situation in which the running multi-programs do not show too much locality and the swapping activities are close to thrashing. The latter frequently occurs when a bunch of small interactive jobs is running in a timesharing system, for example, database information entering center, experiment training lab, etc.

One distinction has to be made before leaving this section. The Write Look-Ahead Buffer is different from the conventional cache in that it never reads a block into its storage as a consequence of a local read miss during the computation of processors. The blocks that are stored in the Write Look-Ahead Buffer are all produced by writing through operations or write misses. They are waiting, at least at its early stage of staying, for writing back to the shared memory. Hence, its function is more like a write multi-buffer rather than a small, fast, first-level memory in a memory hierarchy. That is why, in this article, we insist on using "a Write Look-Ahead Buffer", a lengthy, but a precise, meaningful nomenclature. Loading a read miss block into the look-ahead buffer does not increase the hit ratio. The only benefit of doing this is in the situation in which a read miss from the other processor short time interval wants to share the loaded block in a sufficiently, a considerably rare case. In contrast, loading the read miss blocks will greatly increase the opportunity of the write block collision as the read miss processing could compete for the same buffer block as the write processing. Besides, loading the read miss block to the write buffer itself increases the bus traffic between the memory and the look-ahead buffer, resulting in either prolonging the writing back time or complicating the look-ahead buffer controller, quite possibly degrading overall performance.

4.3. The WLAB Cache Coherence Protocol

In the previous section, we already, at least partially, mentioned, the cache coherence protocol for this WLAB cache organization, when we discussed the configuration of the Write Look-Ahead Buffer and the private caches of the local processors using a single shared memory. To expound upon this idea, in this section, we present a summary of the protocol based on the proposed system. In fact, there exist many strategy choices for said protocol. One which we describe here stands for one of these strategies.

The protocol is a snooping one if people insist in classifying two big cache coherence protocols -

directory based and snooping protocols [HenP90a]. However, unlike almost all the other snooping-oriented protocols, there are simply two states for each local cache block - valid or dirty if broadcast is used (see the state diagram in Figure 4.5).

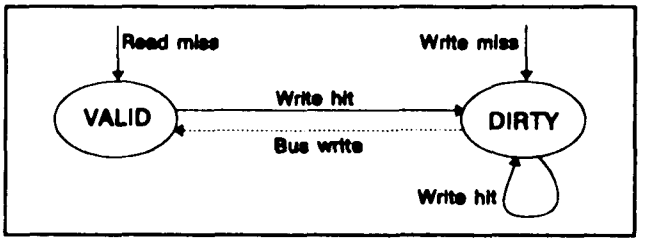

Figure 4.5 State Transition Diagram for the WLAB Cache Coherence Protocol

These two states have the same meaning as in a regular caches. "Valid" can be interpreted as "clean," which states that the contents of this physical cache block is consistent to the corresponding copy in the main memory, while "dirty" refers to the idea at least one word of this block has been rewritten after its last loading. Both "valid" and "dirty" blocks can be either shared or not shared. Broadcast ensures that shared "dirty" block are consistent with one another. When one shared block is going to be "dirtied" by a write operation, other shared blocks will also be updated by it via broadcast. Snooping circuits of local caches are able to capture the write signal and address on the bus, match it with their present "valid" or "dirty" blocks, and thus determine whether there is a shared block in their own region and update it should there be. However, without broadcast, the shared block will be set to invalid. We need not worry that the invalidation of the shared blocks will seriously hurt the performance of the other computing parties. The dirty block would stay in the WLAB for a while. Consequently, the immediately following accesses to those invalid shared blocks are expected to hit the WLAB where it has the up-to-date block. It is a fast buffer access instead of main memory which is extremely slow because the read also must wait for the completion of writing dirty data.

Notice that operations of the protocol in this article are categorized by the conventional terms that were proposed by Smith [Smi82]. That is, the memory accesses are classified into four major operations - read hit, read miss, write hit and write miss. Here is the protocol:

Read Hit - It requires no special processing. It will be responded immediately like most

cache coherence protocols. However, the read hit ratio is supposed to be increased as a result of the elimination of reading disabled shared cache blocks when broadcasting write-through technique is embraced. This is because no active cache blocks are swapped out of the cache due to data inconsistency caused when one of the shared blocks is polluted.

Read Miss - The requested miss block is looked up simultaneously in the WLAB, other local caches and the global shared memory. If a match occurs in the WLAB, the most recent copy of the block is loaded directly from the buffer to the local cache of processor. It is a preferable operation because it reduces the read miss latency by one cache cycle delay instead of one or two memory cycle delay. If no match happens in the look-ahead buffer, the block is read from the memory since now only the single shared main memory has the most recent copy. The memory access takes place simultaneously with the access to the look-ahead buffer, so no extra cycle will be lost when the access to the look-ahead buffer fails. It need not look for the shared dirty blocks in the other local caches forcing them to be written back first before it can be read to the requesting cache as most write back snooping protocols which costs a lot of memory cycles and adds the bus traffic.

Write Hit - Data and identifiers of the block (it may be considered as the address of the block, but prefer to call the identifier as a simple and trivial address concatenation or translation which may be required to obtain a right location for the global main memory or the Write Look-Ahead Buffer) are quickly written through to the buffer at the same speed as written to the local cache. Meanwhile, the other local snooping cache controllers that listen to the bus traffic catch the block identifier on the bus and try to pinpoint a match within their own local blocks. If a match is discovered in a local cache, either the block is set to invalid or the data is switched from the bus to its matched local block, depending upon policies selected (We use the former for simplicity). The regular local cache access may be delayed, if there exists one local access request, by one cache cycle for refreshing status of the shared local cache block. If no match is located in a local

cache, the regular local cache access goes on without any delay since the two matching circuits work independently and concurrently.

Write Miss - Actually, it is a mixture of the read miss and write hit operations. It broadcasts the identifier or/and data of the block to the Write Look-Ahead Buffer and all the other local caches as the same processing for the write hit. But, before the broadcast, the block must be first read from the Write Look-Ahead Buffer or the shared memory, like a read miss operation. Then the block is updated and loaded into the requesting local cache. At the same time, the snooping and switching, if necessary, take place in all the other local caches. Of course, the write miss will cause the longest latency. However, in contrast to the other protocols, since the block is very possibly to be read quickly from the look-ahead buffer and the updating is performed at the buffer speed - not the main memory speed - the write miss penalty is relatively lighter than that in the other protocols.

Most snooping protocols require at least three to four states, two typical state diagrams for these ones are drawn in Figure 4.6. Comparing Figure 4.5 to it, by keeping only two regular states, we

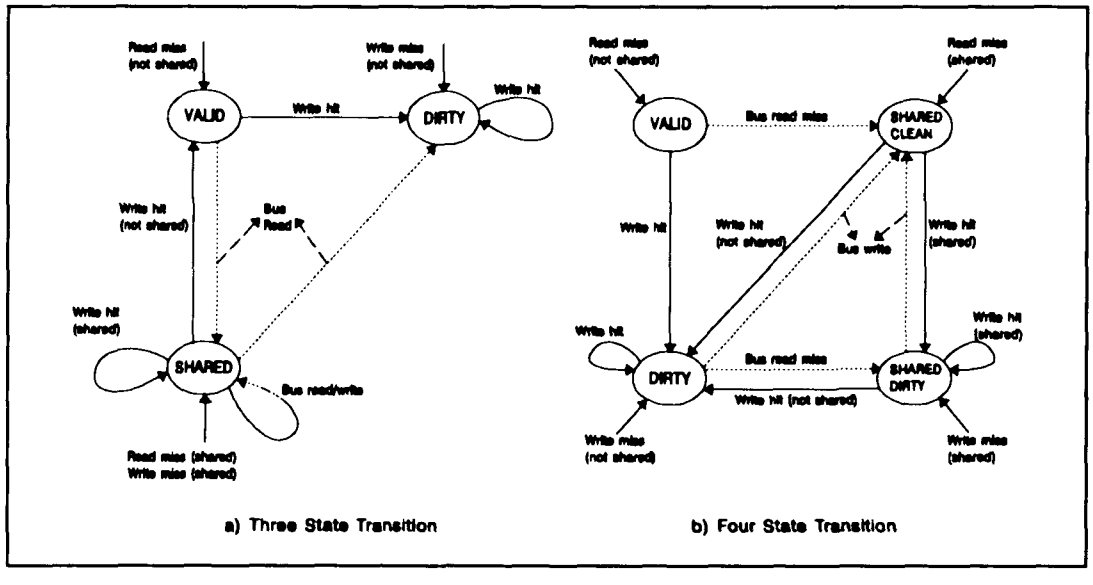

Figure 4.6 Typical State Transition Diagrams for Snooping Protocols

do not need special circuits to test state and make decisions for appropriate transition. Since they are normal, the common cache controller can manage the block accesses. Hence, the protocol functionally reduces the complexity of the local cache controllers by removing the circuit unit for the state testing and decision making. It will raise the rate of the cache processing cycle and thus speed up the local processing. Thanks to the Write Look-Ahead Buffer, it guarantees quick reload of the freshest shared blocks upon demand, saves the possible performance loss, and simplifies snooping manipulation.

4.4. Cost and Performance Expectation

In Section 4.2 where the configurations of the WLAB were discussed, we mainly concerned ourselves with mapping simplicity and convenience. Here we want to address some cost and performance expectations before presenting a real analysis based on the results of our trace simulations by means of PAISLey specification for this WLAB cache memory structure. It delivers some ideas from our original design consideration and motivation.

While the original concept of introducing a global Write Look-Ahead Buffer to the shared memory multiprocessor system is to reduce the write miss latency and cut down the complexity of the cache coherence problem caused by a plain write multi-buffer, we fortunately discover that it can additionally reduce the read miss latency as some missing blocks are read directly from the Write Look-Ahead Buffer, since it is at the cache speed rather than the memory speed. Also, the probability of reading missing blocks from the look-ahead buffer is not low due to the locality of references if there is a burst of writing operations. Moreover, we can increase this possibility by purposely leaving the blocks in the Write Look-Ahead Buffer even after they are already written back to the shared memory. The contents of blocks will disappear only when they are over-written, which is determined by the coming write through operations and the replacement policies. The address queue in the control circuit of the WLAB identifies which blocks or words have not

yet been written back. One bit, attached to each physical block of the WLAB and working as a valid/invalid indicator, can easily determine whether the block is ready to be written over or not. The tag field of the WLAB tells whether it is the right one for reading. It is a more significant result, at least to some degree, than to reduce only the write penalty. It is because the majority of the memory accesses are read operations, and then, certainly, the most misses are the read misses. Therefore, decreasing the miss read penalty is akin to gaining an improvement on the majority operations.

Hit ratio is also expected to be increased by the cache coherence protocol using write thorough with broadcasting strategy. Since every existing cache block, regardless of its sharing or private status, is valid, the read or write misses caused by the access to the invalid, or dirty, shared blocks are eliminated. No shared blocks will become invalid after writing data to one of those shared blocks in the local caches if broadcasting takes place, which spends only one cache cycle delay for all involved parties. Hence it reduces the chance of the read miss and increases the block shareability. The block will become non-sharable after one write access happens to the block in all the protocols using the write back policy, which seems a popular policy among the local caches for single processor, or a write through policy without broadcasting. The block must be re-read if other processors share and request it in any following processing. One may argue that the improvements come from the write through policy that could probably increase the bus traffic. By observing the facts that the increase of the hit ratio reduces bus traffic, and read operation is the most frequent memory accessing operation and write operation is comparatively infrequent with regards to the read operation, this could not become a serious question. Successive gains in the read and write hits pay off for the slight loss in write through processing. The loss can be additionally slimmed down to a minimum by buffering the write through words or blocks, temporarily, in the WLAB without blocking the regular processing speed. Archibald and Baer's evaluations [ArcB86] have already shown, in many cases, the superiority of two protocols - Dragon and Firefly - that apply the write through protocol to the shared blocks. Those protocols do not even utilize the Write Look-Ahead Buffers, which make the write through operation to be

realized at almost the same speed as that of the local caches. It is equivalent to say that the existence of the Write Look-Ahead Buffer greatly increases the memory bus bandwidth and will offset the write through penalty. Those analyses make us believe that the gain will significantly surpass this little loss. In addition, the function of the hardwired local semaphores and local testing technique will also reduce the bus traffic for the synchronizing purpose.

We have already mentioned that the rate of both the global WLAB cycles and the local processing cycles of the private processor can be potentially increased due to the simplicity of the coherence protocol. Only two normal, valid and dirty states, as in the regular cache systems are allowed here with no any other state's expansion. Nevertheless, most snooping cache coherence protocols have three or four states. These new states are introduced merely for coherence detecting purposes. For every local cache access (excluding read hit), the cache controller has to test the state of the currently visiting block and then decide a transition, that is, a proper process that complicates the implementation of the cache controller and certainly adds more time to process.

Almost all the benefits we have deliberated so far are related to the Write Look-Ahead Buffer. Still, some questions remain: how large should the look-ahead buffer be? Is it possible that the new coming write through block will over write the block that is to be written back to the memory but has not done so yet? We call the latter problem the WLAB collision. Note that the occurrence of this collision is much less frequent than that in single buffer or double buffer circumstances. The former question is complicated by the real multiprocessor environment, but its relatively optimal size can be obtained from simulations. We recommend a WLAB with sufficient size for better performance. That is, choose a WLAB which has a size is a little bit bigger than necessary. It would not cost a lot due to inexpensive hardware chips and rapid development of modern integrated circuit techniques.

The frequency of the WLAB collision is related to the size of the Write Look-Ahead Buffer, the number of multiprocessors, etc. A large WLAB will definitely reduce the probability of collision.

Actually, a well-organized look-ahead buffer will reduce the write buffer collision close to nil. Remembering the fact that the write access itself is the minority operation during program running, zipping the Write Look-Ahead Buffer collision is not impossible in reality, although, theoretically, it is always going to happen.

Even when collision does occur, it is not difficult to resolve. One of the conflict blocks already in the look-ahead buffer will be selected and forced to be written back promptly to the shared memory, and leave the room to the new arrival block. This processing may cost one or two memory cycles delay for the party who caused the collision. But, as it is a really rare case for the proper size of a well-organized Write Look-Ahead Buffer, it is not expected to affect the overall performance.

We do not expect that a lot of the blocks will stay in the Write Look-Ahead Buffer and wait for writing back to the shared memory. This situation would induce a lot of the WLAB collisions and degrade performance. Thanks to the high local hit ratio and comparatively infrequent write access, we can easily keep this expectation in most cases. Nonetheless, if the dispatching ratio of cache and main memory is too big, the situation will be disastrous because the colliding effects are cascaded and accumulated due to a very slow memory delay and fast cache processing. Because of contemporary semiconductor main memory, the speed dispatch of cache and main memory is not going to be big. However, an increasing number of multiprocessors is equivalent to increasing the speed dispatch in the sense that all processors are contending for the same shared memory. This is another main reason to select a relative large multi-way and multi-set WLAB. A large WLAB does not just ease the Write Look-Ahead Buffer collision, but also reduces the miss latency because more blocks could be read directly from the Write Look-Ahead Buffer. As the cost of hardware chips continues to drop dramatically, the cost of an additional Write Look-Ahead Buffer becomes relatively cheap compared to a high performance processor. One has no reason to sacrifice performance just for saving pennies after one has already paid dearly for high performance multiprocessors.

The last open problem is that the FIFO structure of the address queue in the WLAB control unit that keeps the order of block writing back to the shared memory is the same as the order of the arriving blocks. If there exist, from time to time, a certain number of blocks stored in the Write Look-Ahead Buffer waiting for writing back, we might not need to keep the arrival order and may adopt some other structure - like the Least-Recently-Used strategy - to improve the buffer hit ratio when local misses occur. It is a very interesting problem and we will see the results of simulation.

5. SPECIFICATION AND PERFORMANCE OF THE WLAB CACHE SYSTEMS

5.1. The PAISLey specification of a WLAB system

The PAISLey specification of the Write Look-Ahead Buffer is straightforward described in a top-down method. The main features of this specification will be addressed in this section. Without delving into every detail of the complete specification, as such a discussion would become lengthy and unnecessary to the essence of this paper. A complete version of the PAISLey specification for the WLAB system of four processors is printed in Appendix B. Several functions are renamed or simplified here for the purpose of concise illustration. Furthermore, in order to shorten the length, all the declarations for functional domains and ranges are omitted in this section though they are used, in fact, as we recommend and can be found in their complete specification of Appendix B.

The local caches and the global Write Look-Ahead Buffer are of similar structure, shown in Figure 5.1, except that the latter has an address queue to record the address to be written back or through, depending on which policy is applied. They are declared as

CACHE = #1.. NumberOfSets < * SET > ;

WLAB = ADDR-QUEUE*BUFFER ;

BUFFER = #1.. NumSetInBuf < * B-SET > ;

where each set in cache or write buffer is composed of valid bit, tag and block of data, that is

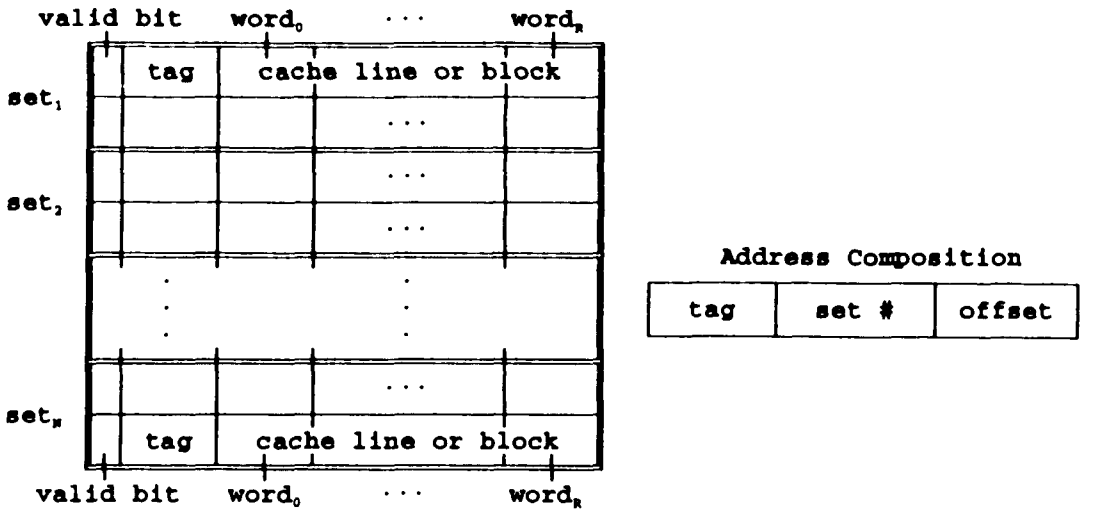

Figure 5.1 A K-Way Set-Associative Mapping Cache Organization (K=2)

SET = VALID-BIT*TAG*LINE

"VALID-BIT" is either 0 or 1 as the name implies, with 0 meaning valid and 1 invalid. "TAG" can be any integer within the address space. We leave "LINE," which is called "block" in some other literature, unspecified since we only concern ourselves with references, that is, locations of data, but not the data themselves. The shared memory bus and local caches are declared as asynchronously autonomous processes whose interactions are realized now by sharing the common Write Look-Ahead Buffer. The system declaration is specified as follows

```
( memory-bus-cycle [ initial ] ,
  global-semaphore-cycle [ 0 ] ,
  p#1.. NumOfProcessors
        < , cache-p-cycle [(initial-counter-p, initial-cache-p)] >
) ;
```

Recall that *p#1.. NumOfProcessors* < ··· > is just a repetition structure. Accordingly, clause *cache-p-cycle[(···)]* is duplicated "*NumOfProcessors*" times with p replaced by 1, 2, ···, *NumOfProcessors* and those duplicating functions are separated by a comma. The formal parameters "*initial*," "*initial-counter-p*," and "*initial-cache-p*" are all mapping functions used to initialize the contents of the Write Look-Ahead Buffer and local caches. By implementing different initializing functions, we are able to realize either cold or warm starting cache simulation. Since multiple processors compete for the access right to the single shared Write Look-Ahead Buffer, a synchronization mechanism such as a semaphore similar to the one addressed in Section 3.2.3 is employed here to ensure the exclusive accesses. Its function and specification is quite close to Specification 7 of Figure 3.14, but the mapping definitions and declarations are nested all over the WLAB specification in which they are required. We will see in the subsequent paragraphs.

"*Memory-bus-cycle*" describes the process that writes the data in the Write Look-Ahead Buffer to the shared main memory. It tests if there is a dirty word or block to be written through by checking whether the address queue is empty or not. Where there is a word or block, it is written to the main memory according to the location in front of the address queue. The statement could be

```
memory-bus-cycle [ (memory, wlab) ] =
        / is-wlab-empty:  (memory, wlab) ,
          True:  write-word-to-memory [ (memory, wlab) ]
        / ;
```

To accomplish a simulating task, the actual work of function "*write-word-to-memory*" in this operational specification is simply a "dequeue" operation to the address queue in the WLAB control interface to main memory. The process "*global-semaphore-cycle*" is responsible for the control of the exclusive access to the Write Look-Ahead Buffer. It maintains a binary semaphore for synchronization. Initially, the semaphore has value zero since no access ever begins. Here

is its definition

```
global-semaphore-cycle [ semaphore ] = send-signal [ semaphore ] ;
```

Processes "*cache-p-cycle*", where p equals 1, 2, ···, until the genuine number of the processors, specifies the structures and coherence protocols of the private caches of local processors as we mentioned earlier in this text. A local processor cycle starts at reference fetching, address mapping, and then performs read or write, hit or miss operations according to the real situation. The initialization of the local cache varies, depending on whether a cold or warm start. The p-th cache cycle is defined as

```
cache-p-cycle [ (counter-p, cache-p) ] =
        ( increment [ counter-p ] ,
          address-map-p [ ( next-reference-p [ counter-p ], cache-p ) ]
        ) ;
```

where "*counter-p*" is used to count the number of fetches, a parameter for statistical measurements. Function "*address-map-p*" maps the address obtained by evaluating function "*next-reference-p*" to tag, set number and word number in a local "*cache-p*," and then decides if it is a hit or miss operation via the associative searching. Therefore, it has the form of

```
address-map-p[ (address-p, cache-p) ] =
    / equal[(associative-map-p[(address-p, cache-p)], Hit)]:  hit-processing-p[ ( ... ) ]
      True:  miss-processing-p[ ( ... ) ]
    / ;
```

Both hit and miss operations must be further broken down into read and write operations. To save context space, only the specification of hit process is going to be demonstrated here. For

simplicity, we do not differentiate instruction fetch from data read since only one cache is used in each local processor for both instruction and data. However, we do take account the difference in our specification for trace simulation. The definition for the hit process follows:

```
hit-processing-p [ (address-p, rw-p, cache-p) ] =
        / equal [ (rw-p, R) ] :   read-hit-p [ ( ... ) ] ,
          True :                  write-hit-p [ ( ... ) ]
        / ;
```

Flag "*rw-p*" indicates whether the current access is a read (R) or write (W) operation. If it is a read one, then read hit occurs; mapping "*read-hit-p*" simply returns the cache itself, since no further operations are of interest for this cache simulation. After read hit, the p-th cache is immediately ready to go into the next fetching cycle. Otherwise, a write hit takes place. The real simulating operation depends on the selected protocol strategy. The write through policy with broadcasting has been chosen to implement the WLAB cache system simulation. Thus, the write hit operation requires that the data be written to the Write Look-Ahead Buffer where it waits for its turn in order to copy back to the main memory. It is specified as

```
write-hit-p [ (address-p, cache-p) ] =
    / equal [ (get-signal-p, 0) ] :   write-through [ (address-p, cache-p) ] ,
      True :                          wait-signal-p [ (address-p, cache-p) ]
    / ;
```

Function "*write-hit-p*" first checks whether some other processor is accessing the Write Look-Ahead Buffer by testing signal value of the semaphore. This is done by evaluating the renamed exchange function "*get-signal-p*". If the exchange primitive does not return to 0, implying that one of the other processor does access the buffer, then the intrinsic function "*equal*" fails and the functions in the "*True*" case are evaluated. Thus no actual write access takes place and the

processor has to wait on the semaphore cleared. If the exchange function does return 0, then function "*equal*" succeeds and function "*write-through*" will be evaluated to updates the Write Look-Ahead Buffer and projects the first item "*cache-p*" for next cache cycle. The subsequent clause describes its definition.

```
write-through [ (address-p, cache-p) ] =
        proj [ ( 1, (cache-p, clear-signal-p [ update-wla-buffer [ ( ··· ) ] ] ,
                    k#1.. NumOfProcessors < ,  broadcast-k [ address-p ] >
                ) ] ;

k#1.. NumOfProcessors
        < ; broadcast-k [ address ] = xm-broadcasting-k [ address ] > ;      "k<>p"
```

Notice that the semaphore will be cleared after the data is written to the Write Look-Ahead Buffer. Indeed, "*get-signal-p*" and "*clear-signal-p*" rename the same exchange channels. The former tests and sets value one to the semaphore whereas the latter clears up the semaphore to release the Write Look-Ahead Buffer. They are specified as follows

```
p#1.. NumOfProcessors
        <  ;  get-signal-p = xr-signal [ 1 ] ;
              clear-signal-p [ (···) ] = xm-signal [ 0 ]
        >  ;

send-signal [ semaphore ] = x-signal [ semaphore ] ;
```

Here function "*send-signal*," invoked by process "*global-semaphore-cycle*," works independently and sends a signal value of the global semaphore to all local processors. Real time channels "*xr-signal*," when triggered, attempt to exchange the lock value with channel "*x-signal*" which

corresponds the status of the Write Look-Ahead Buffer. When an exchange occurs, the semaphore will be set to 1 as the test-and-set operation. Notice that *xr-*, and *xm-* channels cannot match themselves, but can match *x-* channels. Thus it specifies the situation in which many processes compete for a single shared resource. It is a treasury of PAISLey that makes the specification of synchronizing mechanisms for multiple processes so convenient and intuitive.

Back to the specification of "*write-through*," we recognize that it also broadcasts the address of a dirty word to all other processors. The snooping units of the local caches capture this address and search its own contents to look for a match. If a match occurs, the dirty word is switched into the matched location of the local cache, while the regular cache fetching is postponed by one cycle. If not, the cache fetching processes as usually without any delay. Hence, the specification of process "*cache-p-cycle*" should be modified as

```
cache-p-cycle [ (counter-p, cache-p) ] =
        / is-share [ ( get-broadcast-p [ location-p ] , cache-p ) ] :
                ( counter-p, update-shared-word-p [ (location-p, cache-p) ] ) ,
          True: ( increment [ counter-p ] ,
                  address-map-p [ ( next-reference-p [ counter-p ], cache-p ) ]
                )
        / ;

get-broadcast-p [ location-p ] = x-broadcasting-p [ location-p ] ;
```

Since what we are interested are reference activities, not contents of caches for cache simulation, we need not specify function "*update-shared-word-p*." This is because it, in reality, only modifies the contents of the p-th cache at *location-p* but does not change the tag and the other relevant address.

Now, the remaining important matter for operational specifications of the WLAB system is the time relationship among the main memory, Write Look-Ahead Buffer and local caches access. The principal time attributes here are memory and cache access time. It is not difficult to specify the main memory access time, which can be done by simply attaching a time constraint to process "*memory-bus-cycle*," such as one underneath

```
memory-bus-cycle:  ! lb 80.0 ns, ub 80.0 ns ;
```

Nevertheless, we are unable to apply the same principle to processes "*cache-p-cycle*," p=1, 2, ···, *NumOfProcessors* because a local cache cycle process can be postponed by a failure of accessing the unavailable Write Look-Ahead Buffer or updating a dirty shared block. A tight time upper bound may cause an inconsistency while a large one makes no sense. Yet, a lower bound time constraint combined with a bottom-up scheduling mode works, appending the statement below

```
cache-p-cycle:  ! lb 40.0 ns ;
```

The bottom-up scheduling method starts evaluating functions immediately after they are ready and satisfy the relevant time constraints. Consequently, it assures that each cache process is cycled at every 40.0 nanoseconds while no postponement occurs. The time characteristic related to the postponement should be connected individually to the functions which produce or deal with these deferrals. It would be superfluous to dwell on this matter; again, a complete version of this PAISLey specification of the WLAB system with the associative C input utilities and processes is listed in the appendices.

5.2. Simulation and Performance Evaluation

The primary goal of an executable formal specification is to perform automatic system consistency

checking. The objective of computer hardware description aims at providing us with a runnable simulating model. However, as emphasized earlier in this article, an executable specification should be able to collect performance measurements directly, in addition to attesting to internal coherence as it manifests inherent merits or defects of the design itself. All the operational specifications must offer these capabilities, otherwise they lose practical usage. Nevertheless, imperative hardware description languages lack the ability to furnish an instance-free or technology-independent specification that truly reflects design problems not limited by constraints or detours raised by the technique employed. PAISLey, by using applicative high order logic, overcomes those limitations and is capable of presenting a completely implement-independent operating model.

This section exhibits the simulating results gathered from the tracing execution of the PAISLey specification of the WLAB cache organization. The execution of this WLAB cache system specification performs a straightforward trace simulation, a method utilized by most of contemporary cache performance evaluations. The whole simulation is carried on the SUN SPARC II and IPL 4 workstations under the environment of SUNOS Version 4.1.1, a dialect of UNIX system. We adopt, for the address references, the well-known SPEC and ATUM traces developed by Hennessy and Patterson [HenP90a, HenP90b]. Specifically, we have chosen, in our trace simulation, the *dec0*, *forf*, *fsxzz*, *mul2*, *gcc*, *TeX* and *Spice* dinero traces, and have transformed them to the formats recognized by our specification. The last three traces are from the SPEC (System Performance Evaluation Cooperative) benchmarks with a million references of each. It seems no significant difference, from the view of performance measurements, for the number of references used when the number is sufficiently large for each trace. The multiprogramming environment is also deliberated in this specification. During the execution, the simulation can be switched from one reference trace to another via a round robin policy. The number of references for each slice of a trace is determined by either a fixed constant value or a distributively variable one. The degree of multiprogramming is equivalent to the number of traces involved in the simulation. Simulation has been conducted from the PAISLey specification

of the WLAB cache hierarchies for various performance consideration based on uni-processor, dual-processor and quadruple-processor systems. However, in order to shorten the length of context, only most interesting results will be shown in this thesis.

Figure 5.2 displays a fraction of the trace snatched from the execution of the WLAB cache specification in PAISLey as described in the previous section. For the purpose of simple illustration, it keeps track of only two functions at their evaluation terminating points. Each evaluation of the "*associative-map-1*" function returns a binary tuple (x, y), where x belongs to {I, R, W}, standing for Instruction fetch, data Read and data Write, respectively, and y belongs to either "Hit" or "Fail." Accordingly, "*associative-map-1=(R, Hit)*" refers to a read hit occurring with resect to the corresponding local cache. When a read operation fails, the Write Look-Ahead Buffer will be accessed and checked to see whether the referencing block is in it or not. The same principles are applied to both write fail and hit. As a result, the binary tuple (x, y) returned by function "*buf-map-1*" answers this question. Here, "y" could be either "Hit" or "Fail" as before, but with regarding to the Write Look-Ahead Buffer instead of the local cache. "X" could be one of values "Rmiss," "Whit" and "Wmiss,"

Time in 10 ns	1 2 3 4 5
0.00000	associative-map=(R,Fail)
0.00000	buf-map=(Rmis,Fail)
4.00000	associative-map=(I,Fail)
4.00000	buf-map=(Rmis,Fail)
8.00000	associative-map=(R,Fail)
8.00000	buf-map=(Rmis,Fail)
12.00000	associative-map=(R,Hit)
16.00000	associative-map=(R,Hit)
20.00000	associative-map=(R,Fail)
20.00000	buf-map=(Rmis,Fail)
24.00000	associative-map=(R,Hit)
28.00000	associative-map=(R,Hit)
32.00000	associative-map=(I,Fail)
32.00000	buf-map=(Rmis,Fail)
36.00000	associative-map=(W,Fail)
36.00000	buf-map=(Wmis,Fail)
40.00000	associative-map=(W,Hit)
40.00000	buf-map=(Whit,Fail)
44.00000	associative-map=(W,Hit)
44.00000	buf-map=(Whit,Hit)
48.00000	associative-map=(W,Hit)
48.00000	buf-map=(Whit,Hit)
52.00000	associative-map=(W,Hit)
52.00000	buf-map=(Whit,Hit)
56.00000	associative-map=(I,Fail)
56.00000	buf-map=(Rmis,Fail)
... ...	... ...

Figure 5.2 Trace From Execution of the WLAB Cache System Specification

symbolizing that the occurrence of this WLAB associative mapping stems from the Read miss, Write hit or Write miss to the local cache. The miss of instruction fetch is considered in the text as identical to the data read miss, since a unique Write Look-Ahead Buffer is in use. It is not difficult to collect all these measurements from the entire trace of the execution of this PAISLey specification by using either "*cat*" or "*grep*" and relevant UNIX utilities, or any other customer favorable tool.

In order to enforce the argument why a multi write buffer is required in the cache memory organization, we display, in Figure 5.3, a chart of the percentage when two or more words occupy the Write Look-Ahead Buffer derived from our simulation. The percentage is directly proportional to the dispatch ratio between the main memory access time and the local cache access time. The first series is the percent against all cases, and the second is the percent against the cases in which the Write Look-Ahead Buffer is not empty. The high percentage shown by the second series reveals that the write operations usually occur in a short burst and follows the rule of locality.

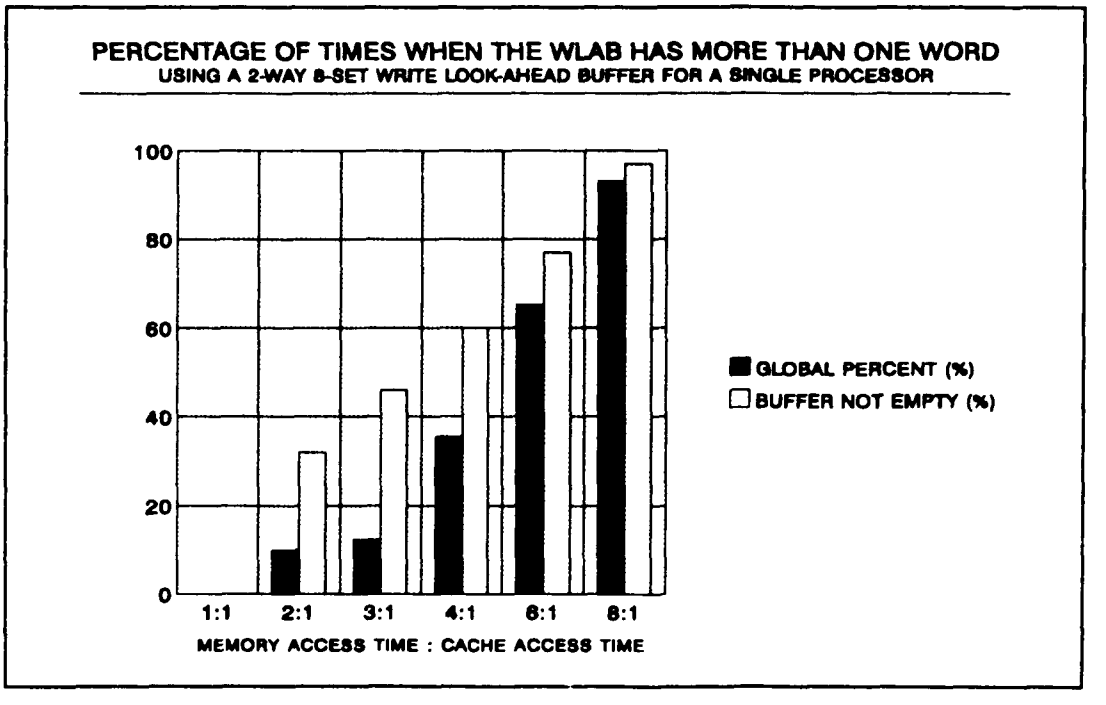

Figure 5.3 Percentage for the WLAB to Have More Than One Word

It makes sense more significantly to use a write multi-buffer. Both series disclose a rapid accumulative effect due to a large dispatch ratio because a large unbalanced speed dispatch could make the buffer grow quickly.

The hit ratios of Figures 5.4-5.8 for a single processor system, Figures 5.9-5.13 for a dual processor system and Figures 5.14-5.18, a quadruple processor system come from running the specification of a 2-way 8-set 256 byte Write Look-Ahead Buffer along with K-way 64-set caches, where K=1, 2, 4, 8, 16 and 32, stretching in the horizontal axis in the figures, representing 1k, 2k, 4k, 8k, 16k and 32k byte caches for a line size of four words and a word size of four bytes. Figures 5.4-5.5, 5.9-10 and 5.14-15 exhibit read and write hit ratio of these caches, distributively. There are two curves in each table, with the lower one representing the regular hit ratio referring to the local cache, the upper one drawn by adding the effect of the Write Look-Ahead Buffer, that is, it is the sum of the local cache hit plus the write buffer hit after the local cache reference fails. One rewarding point which we are eager to indicate is that the Write Look-Ahead Buffer

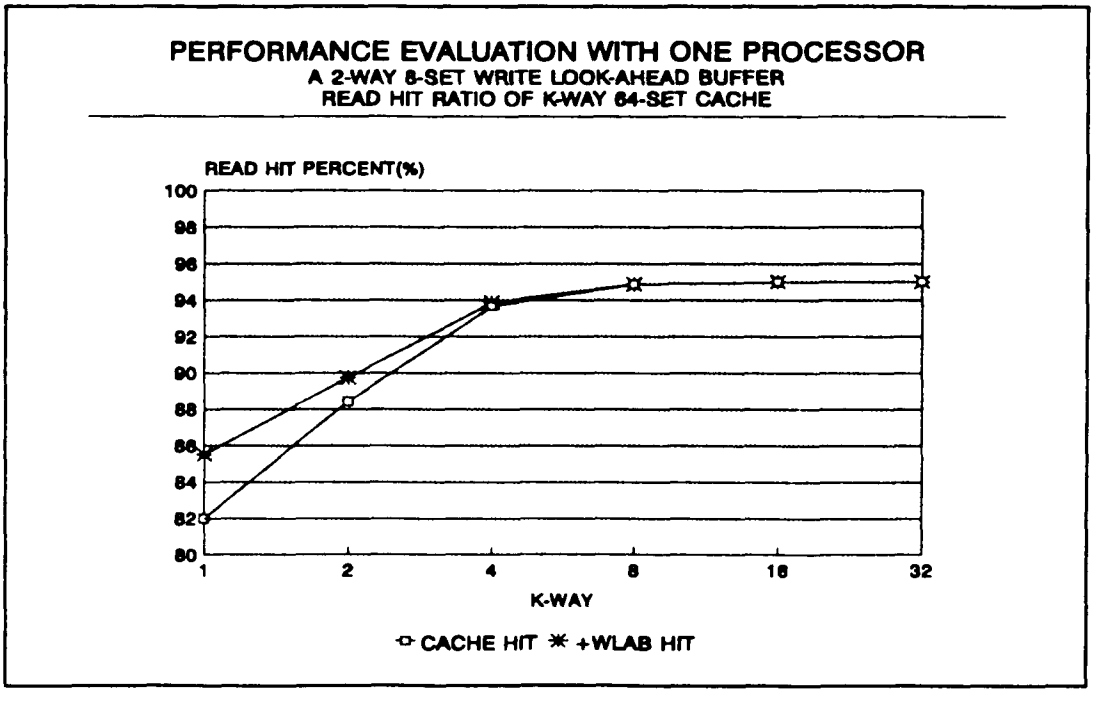

Figure 5.4 Hit Ratio of the WLAB System for Unit-Processor Environment

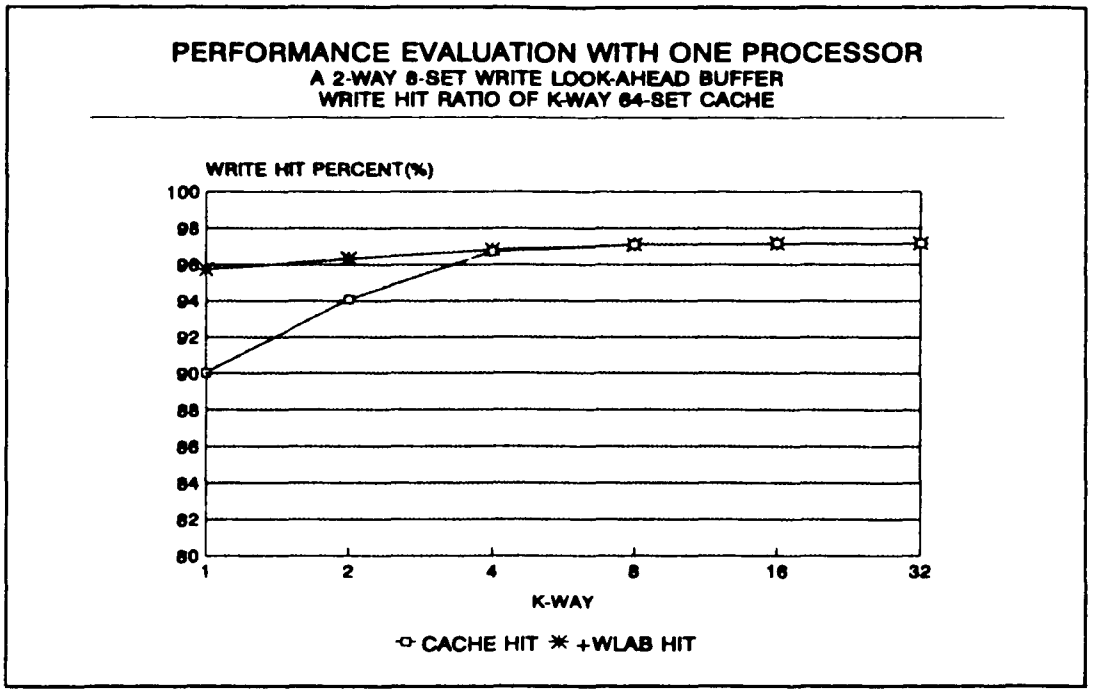

Figure 5.5 Hit Ratio of the WLAB System for Uni-Processor Environment

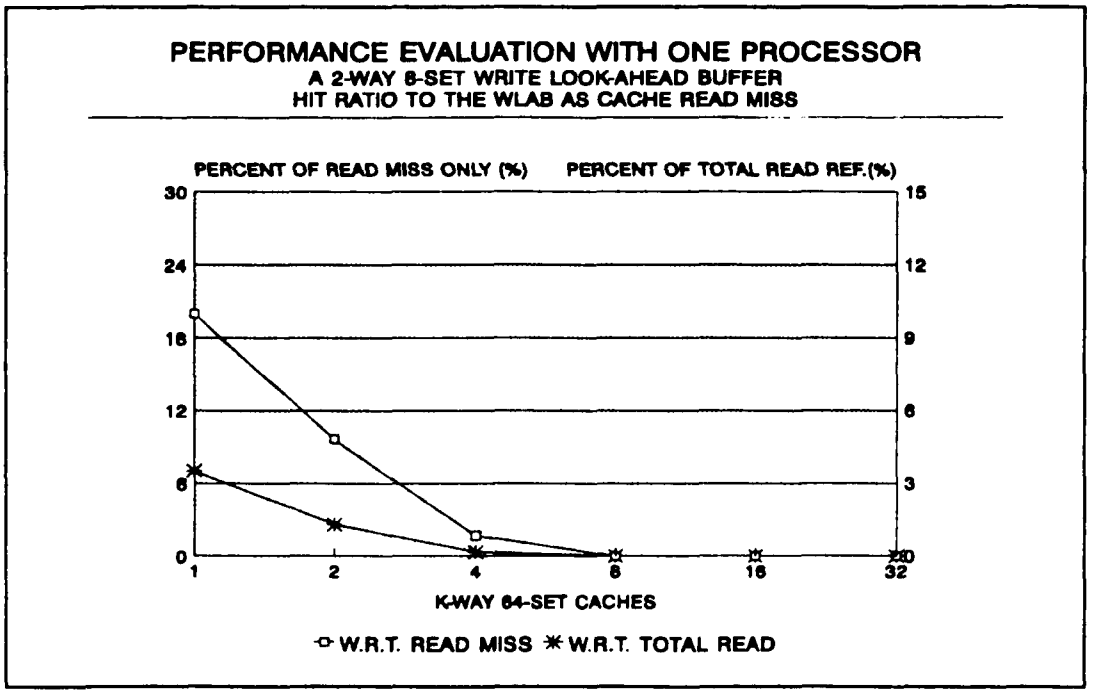

Figure 5.6 Hit Ratio of the WLAB System for Uni-Processor Environment

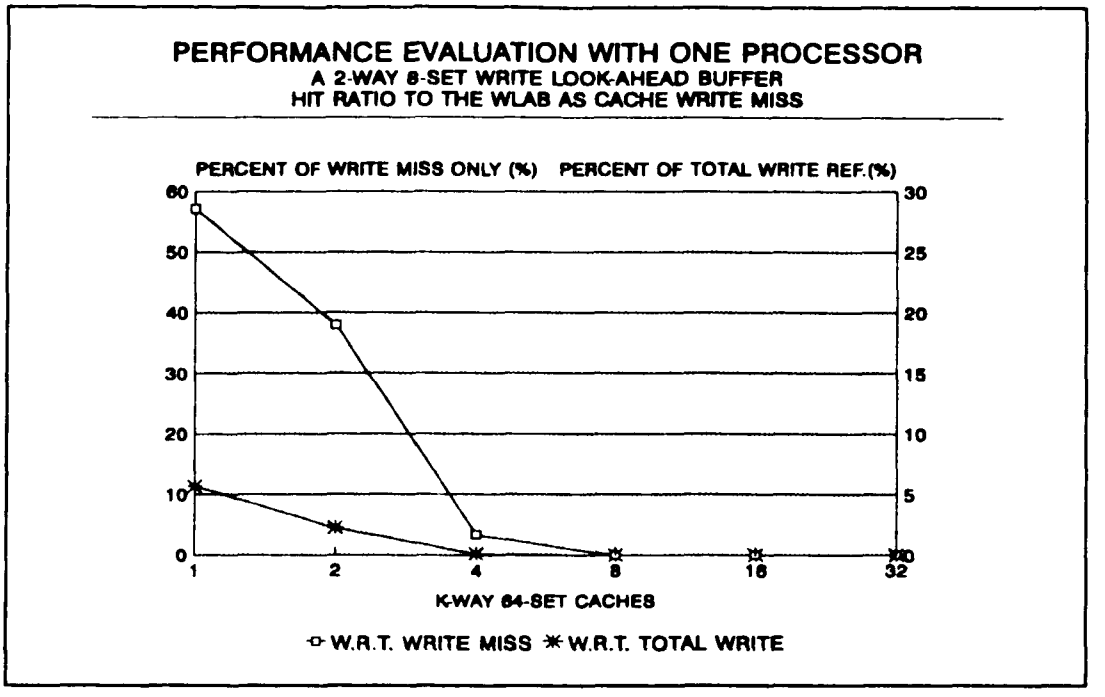

Figure 5.7 Hit Ratio of the WLAB System for Uni-Processor Environment

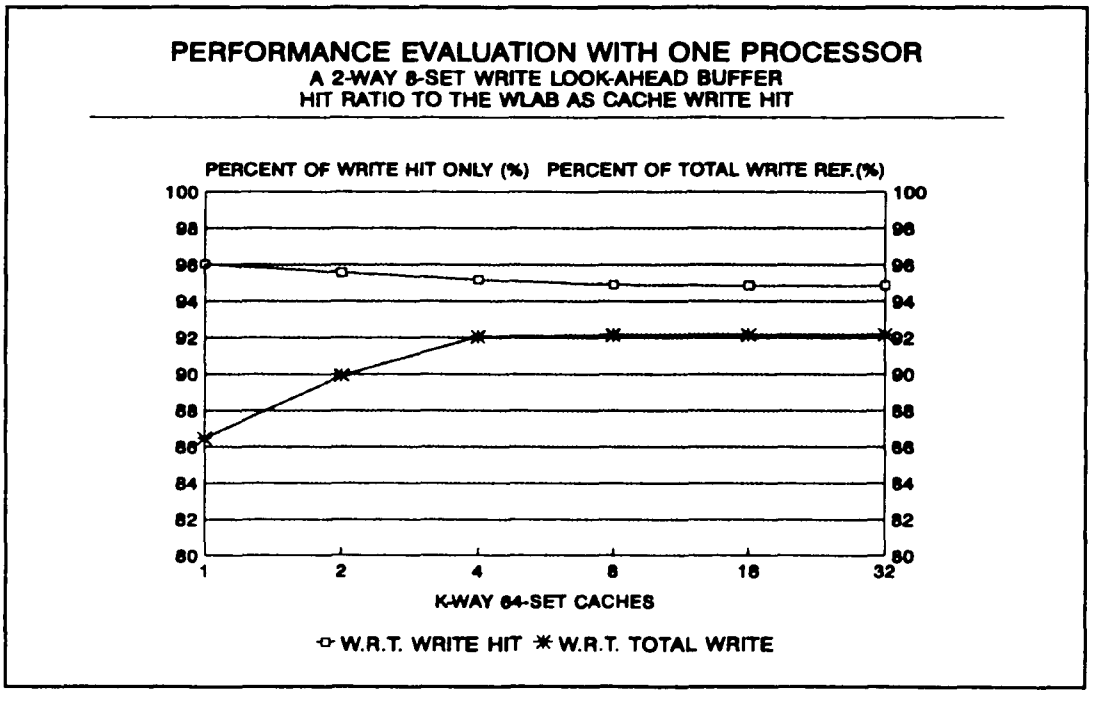

Figure 5.8 Hit Ratio of the WLAB System for Uni-Processor Environment

compensation for the miss operation increases as the hit ratio of the local cache goes down. The upper curves in Figures 5.5, 5.10 and 5.15 are almost horizontal, which ensures a more constant hit ratio. It is grounded on the fact that more opportunities of reading data from the Write Look-Ahead Buffer instead of the main memory are produced when more read misses occur due to a raise of block swapping activities. The phenomenon makes the Write Look-Ahead Buffer more profitable for a large degree of multiprogramming and multiprocessor circumstances where the locality of references cannot be predictable and the hit ratio declines drastically resulting from block thrashing. In this situation, the Write Look-Ahead Buffer functions as a backup cache and is then able to reduce latency by buffering for, and feeding back, blocks to the private caches in the local processors.

Figures 5.6 through 5.8, 5.11 through 5.13, and 5.16 through 5.18 portray the hit ratio of the Write Look-Ahead Buffer after read miss, write hit and miss to the local caches, separately. Each diagram outlines two curves, the lower one representing the buffer hit percentage against the total number of the read or write references with respect to the right vertical scaling axis, and the upper curve for the one against the number of the individual read miss, write hit and write miss references, separately, with respect to the left vertical scaling axis. It is denoted, as we expect, that the Write Look-Ahead Buffer hit ratio is reciprocal to the hit ratio and size of the local cache for both read and write miss operations, see Figures 5.6-5.7, 5.11-5.12, and 5.16-5.17. However, the buffer hit ratio inclines to a constant at a high percentage as depicted in Figures 5.8, 5.13 and 5.18, in which cases the dirty blocks or words are frequently hit themselves in the Write Look-Ahead Buffer. This signifies that many write through operations overwrite to the blocks in the Write Look-Ahead Buffer before it is actually written back to the main memory. In other words, the number of times when the actual writing through takes place to the main memory could be less than the number of times when the write hit takes place and the write through is required without the Write Look-Ahead Buffer. Consequently, it dramatically slims down the memory traffic which is our main concern when we design a cache memory system, especially for a shared-memory system with modern multi-media and multi-processor systems.

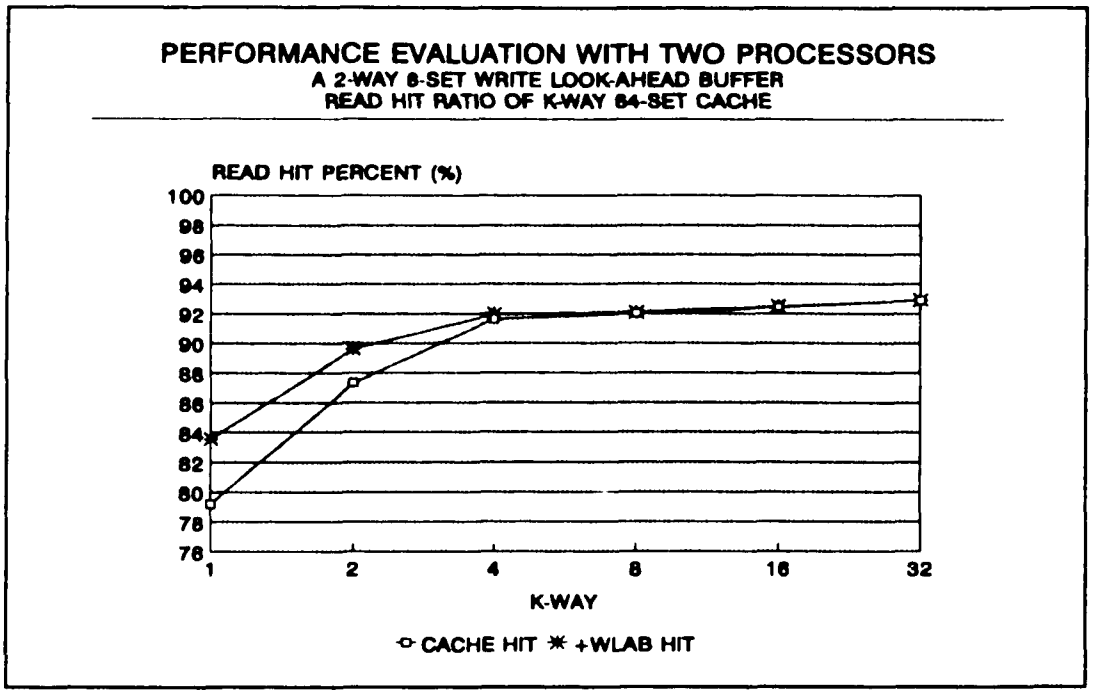

Figure 5.9 Hit Ratio of the WLAB System for Dual-Processor Environment

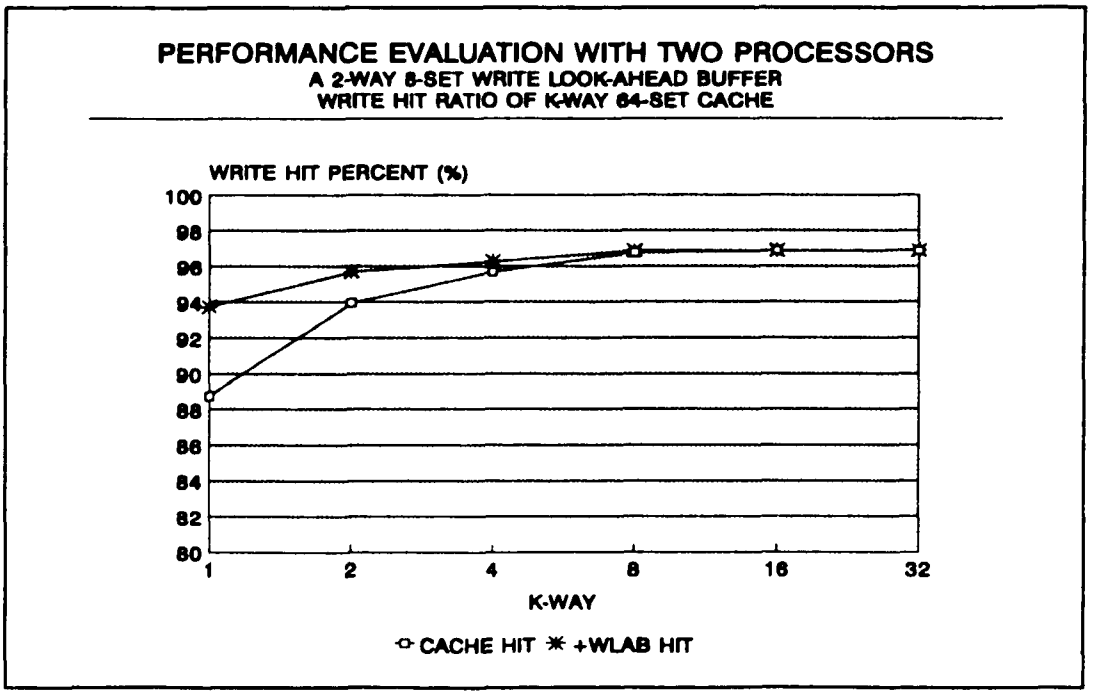

Figure 5.10 Hit Ratio of the WLAB System for Dual-Processor Environment

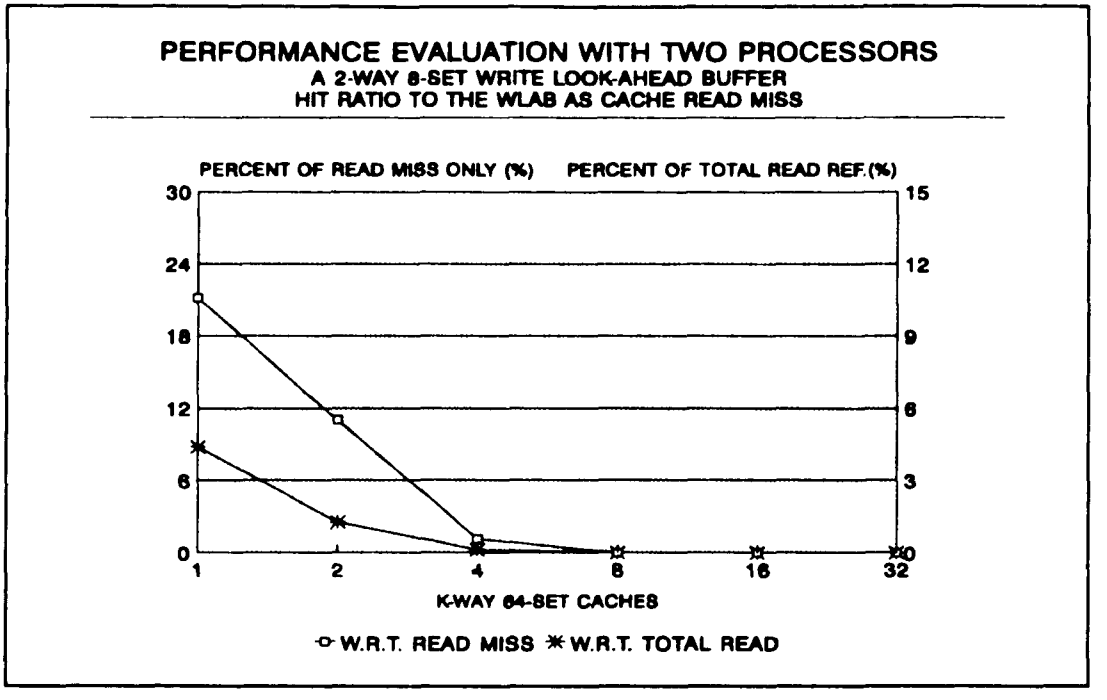

Figure 5.11 Hit Ratio of the WLAB System for Dual-Processor Environment

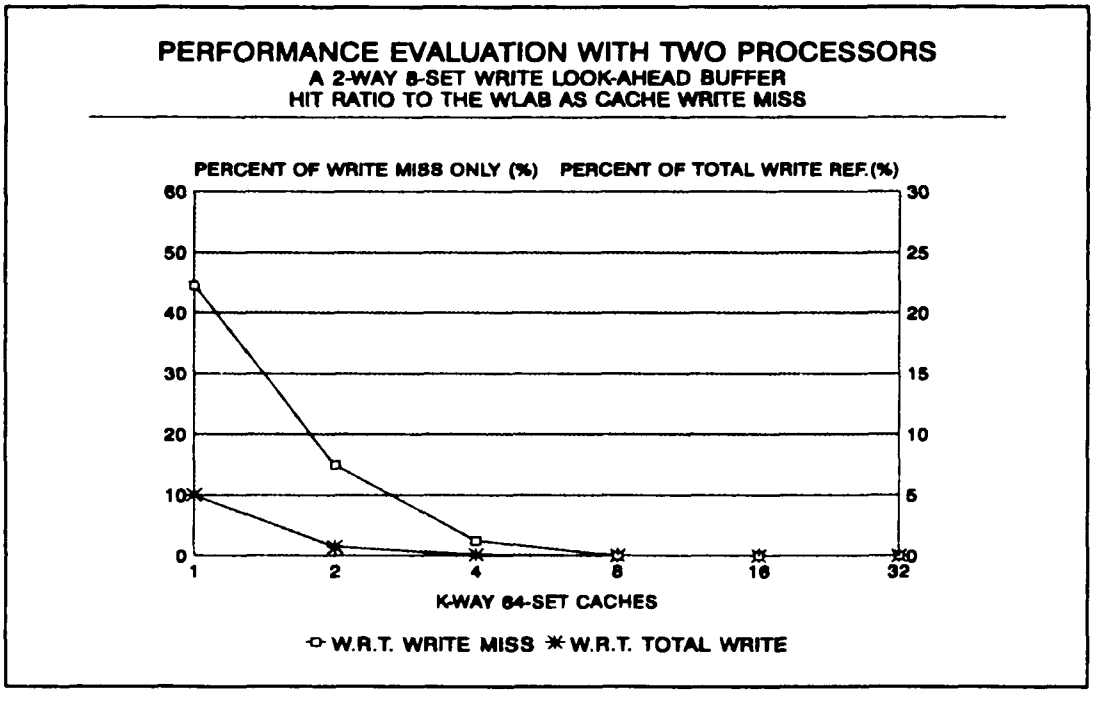

Figure 5.12 Hit Ratio of the WLAB System for Dual-Processor Environment

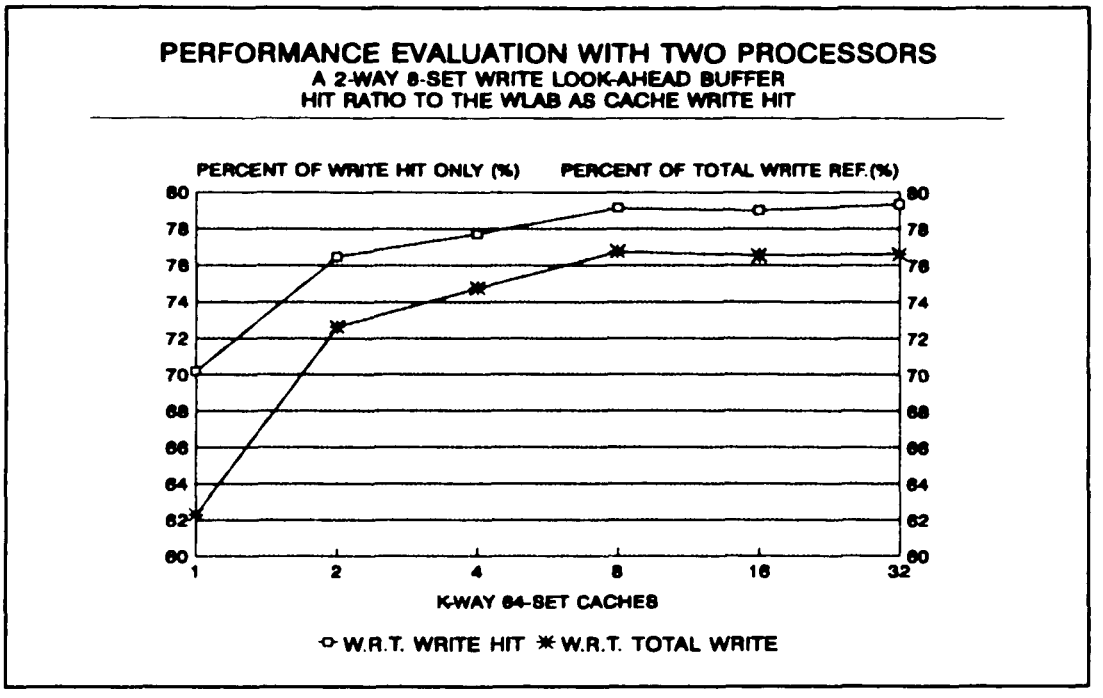

Figure 5.13 Hit Ratio of the WLAB System for Dual-Processor Environment

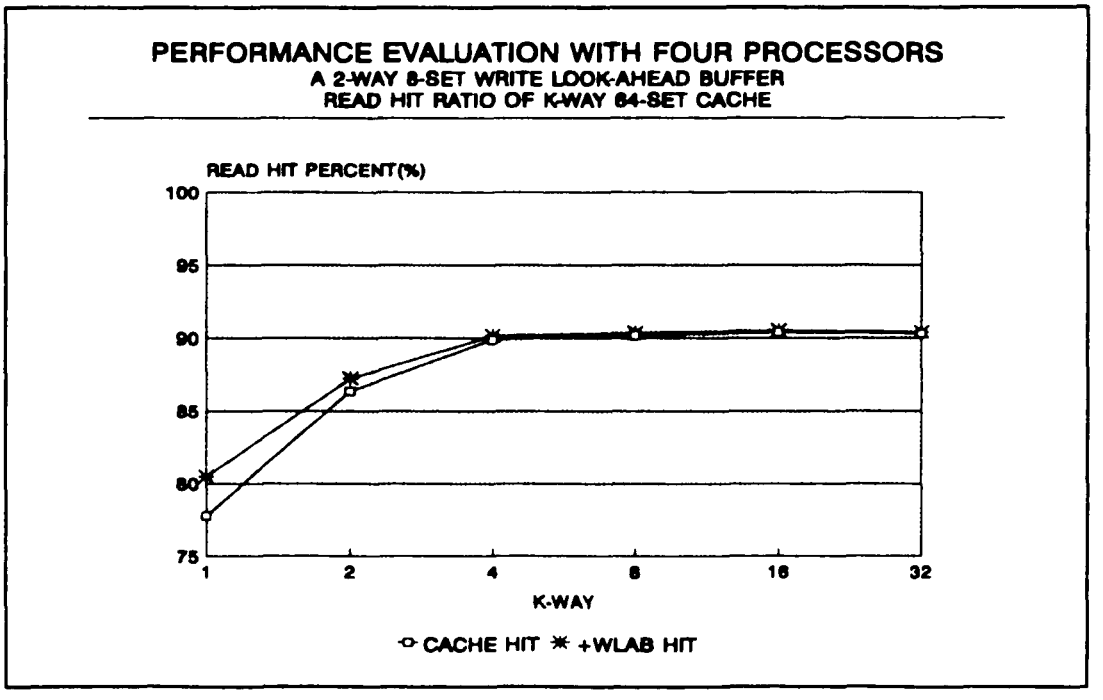

Figure 5.14 Hit Ratio of the WLAB System for Four-Processor Environment

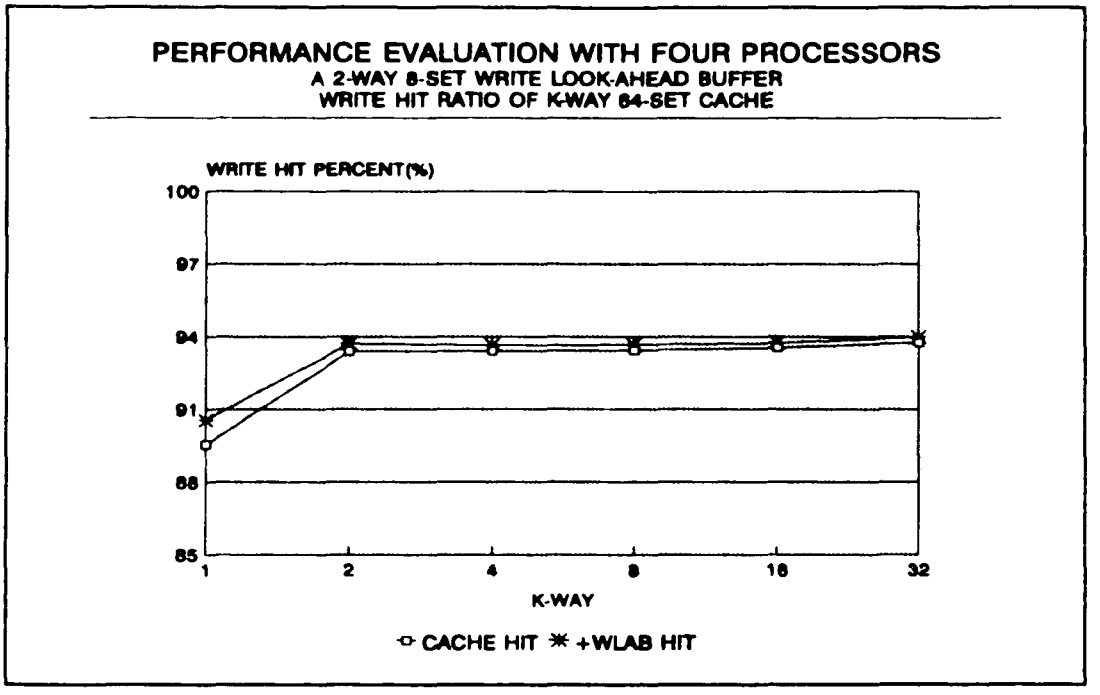

Figure 5.15 Hit Ratio of the WLAB System for Four-Processor Environment

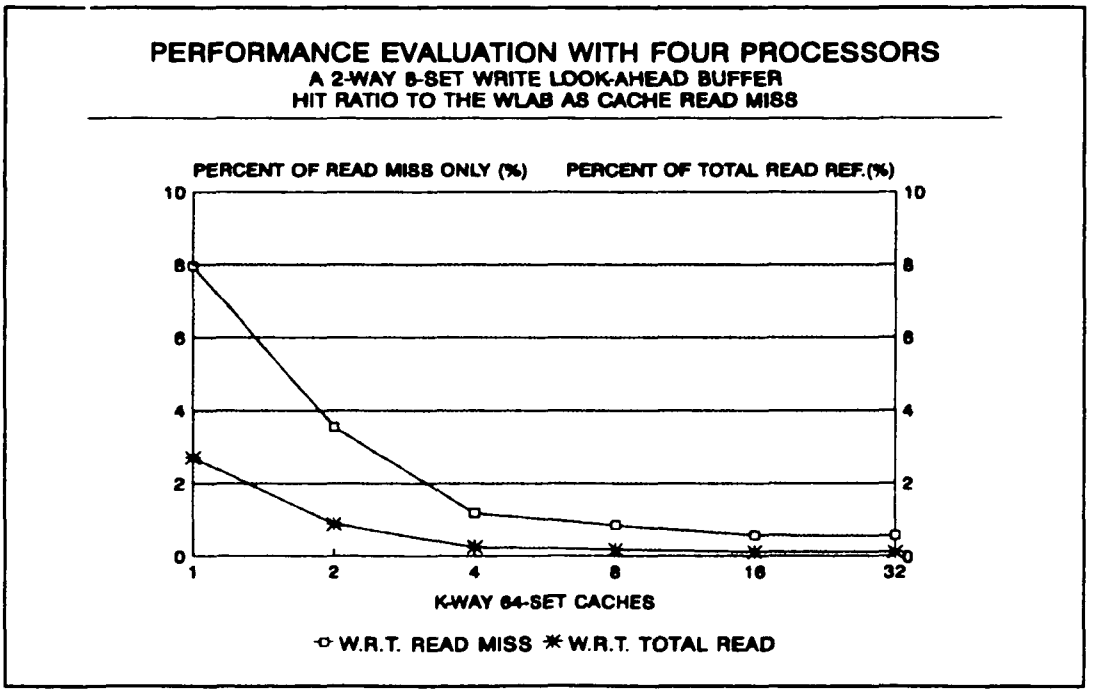

Figure 5.16 Hit Ratio of the WLAB System for Four-Processor Environment

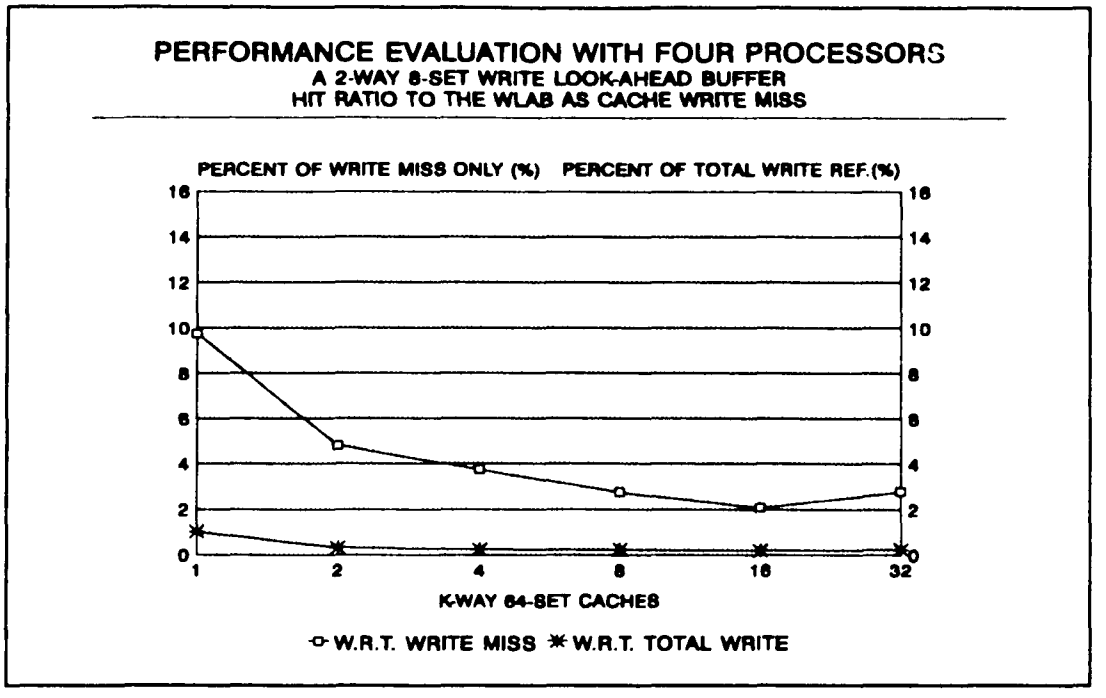

Figure 5.17 Hit Ratio of the WLAB System for Four-Processor Environment

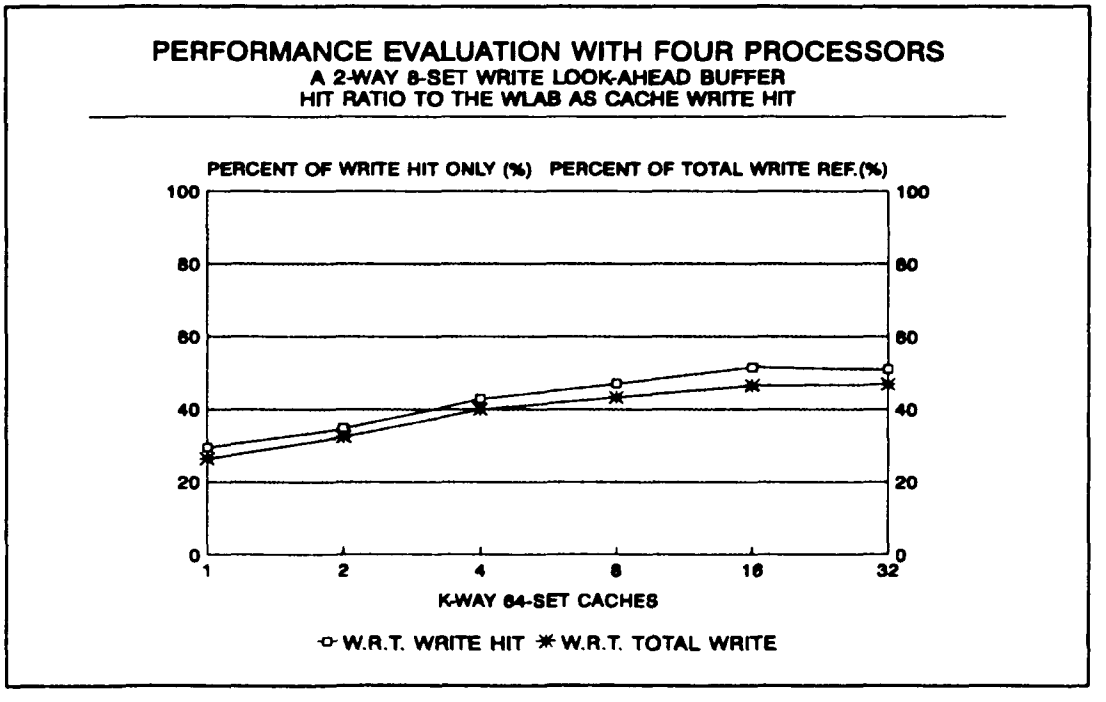

Figure 5.18 Hit Ratio of the WLAB System for Four-Processor Environment

To further illuminate the ability of the WLAB to reduce the memory traffic, we draw the charts for memory traffic ratio and write traffic reduction in Figures 5.19 through 5.23. The memory traffic ratio is defined as a percent of the number of accesses in the WLAB cache-memory instruction traffic to that in the processor-memory instruction traffic if no cache were present [Goo87, MitF90]. We also assume that both the processor-memory bus and cache-memory bus are four bytes wide. Accordingly, loading a block of cache with a line size of four words, for instance, is counted as four memory access cycles. The write traffic reduction is defined as one hundred percent minus the percent of the numbers when the write through operations actually take place against the numbers when the write through would be required without using the Write Look-Ahead Buffer. The memory traffic ratio is just a measurement for a regular cache system, whereas the write traffic reduction is the one derived right away from the merit of the Write Look-Ahead Buffer.

Both memory traffic ration and write traffic reduction are dependent on a variety of factors such

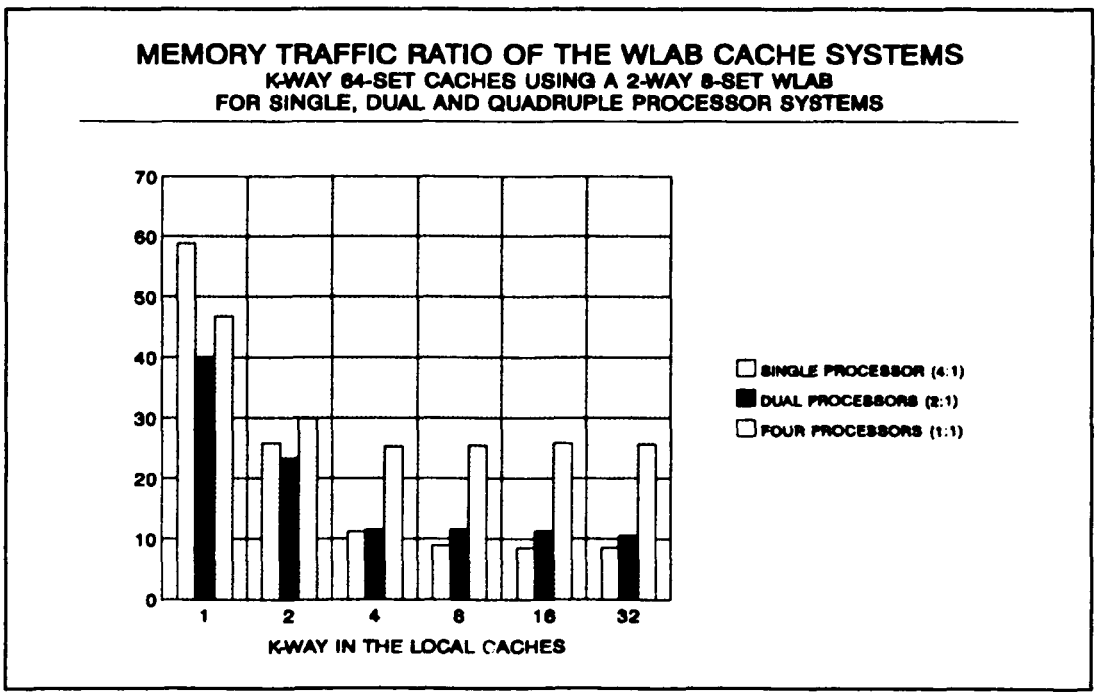

Figure 5.19 Memory Traffic Ratio of the WLAB Cache Systems

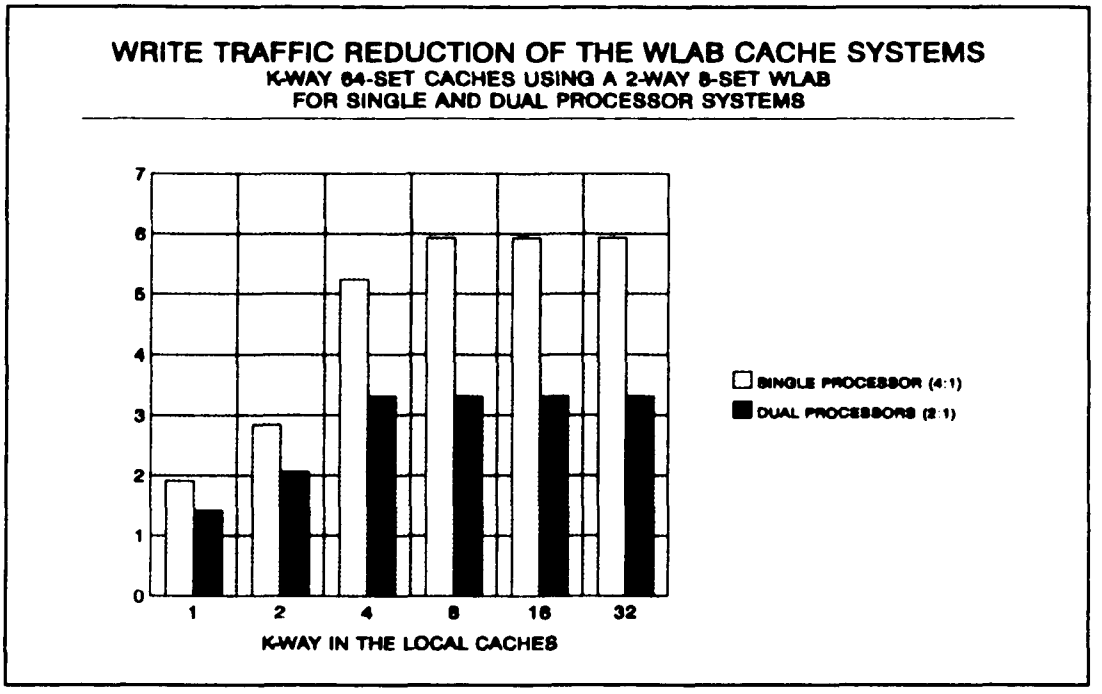

Figure 5.20 Write Traffic Reduction of the WLAB Cache Systems

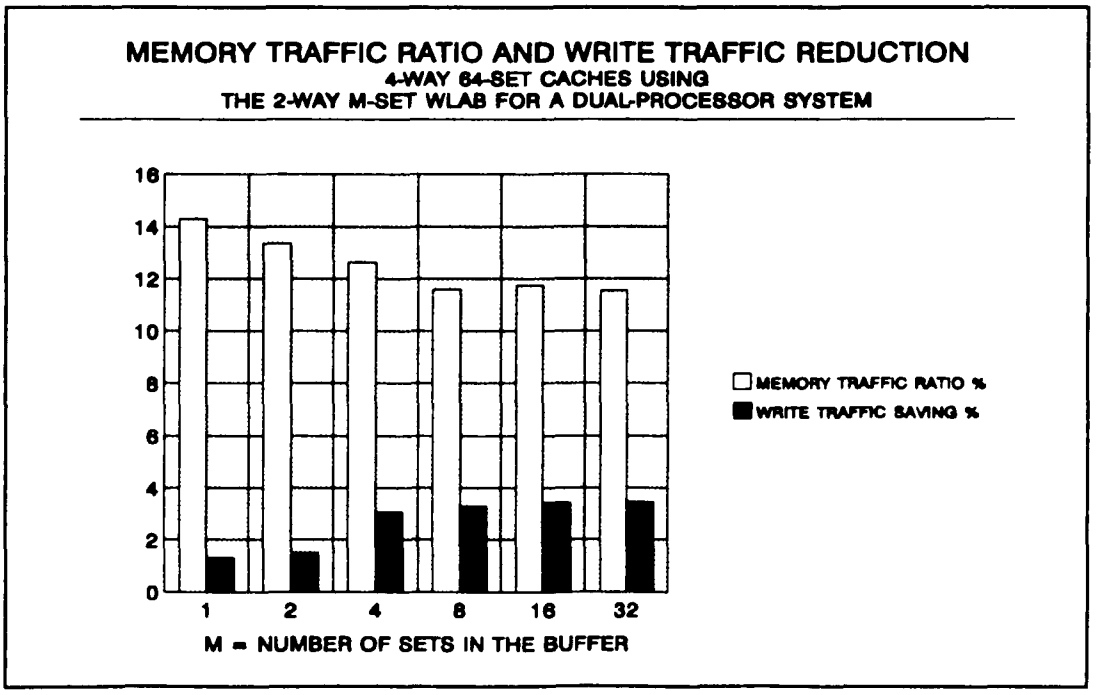

Figure 5.21 Memory Traffic Ratio and Write Traffic Saving for Dual-Processors

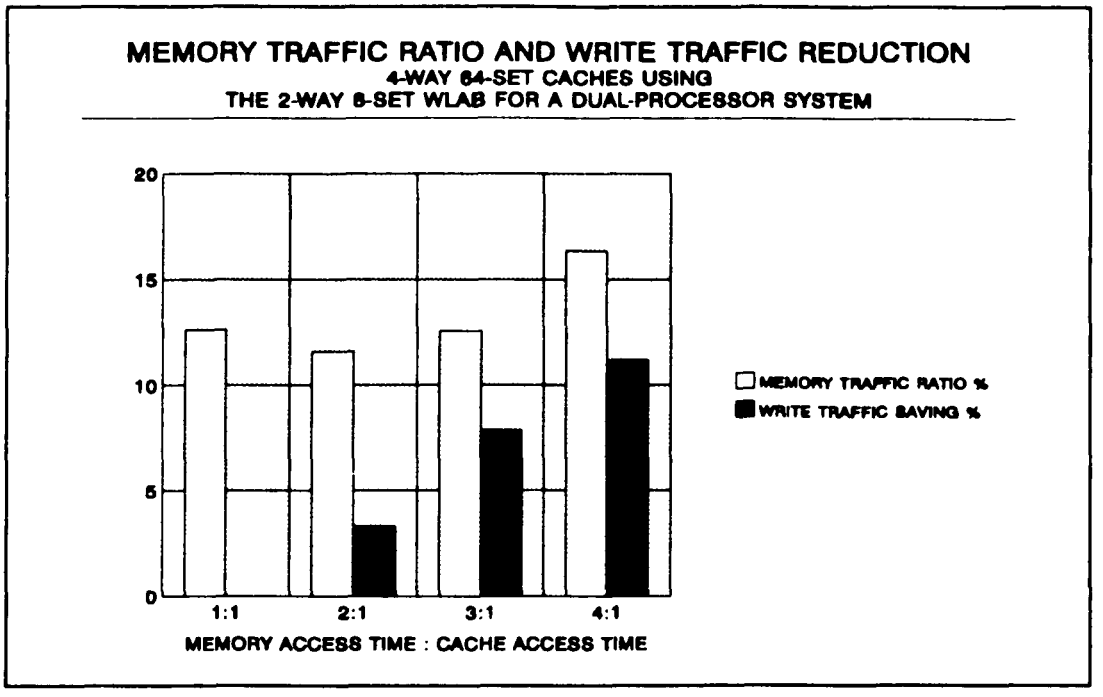

Figure 5.22 Memory Traffic Ratio and Write Traffic Saving for Dual-Processors

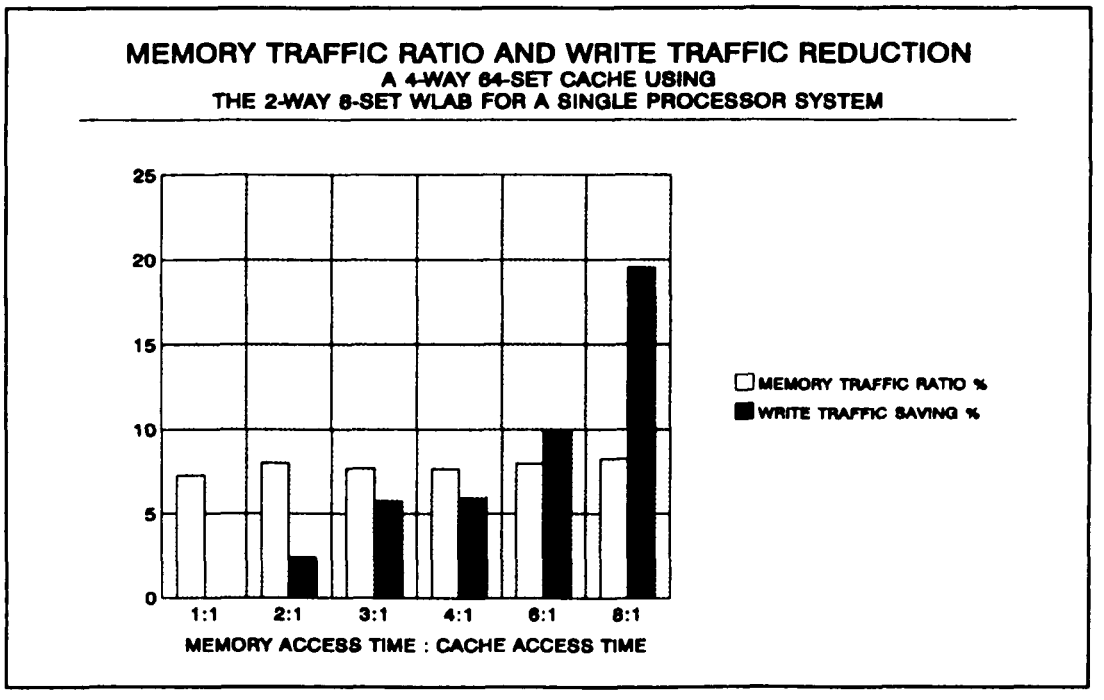

Figure 5.23 Memory Traffic Ratio and Write Traffic Reduction for Uni-Processor

as the characteristic of the trace references, the size of caches, the capacity of the WLAB, the speed dispatch between the shared memory, the private caches in the local processors, and so forth. Many papers [Goo83, 87, MitF90] have been published which discuss the relationship of these factors with the memory traffic ratio. The simulation results of the WLAB cache system, outlined in Figure 5.19 and the first series of Figures 5.21 through 5.23, are in accord with those studies. Therefore, we do not want to rewrite these explanation and analysis redundantly here. However, we do want to dwell on the write traffic reduction more, since it is a distinctive property of the Write Look-Ahead Buffer. From Figure 5.20 and the second series of Figure 5.21, we see that the write traffic reduction is very small when the size of caches and the size of the Write Look-Ahead Buffer is not sufficiently large. This is because the small size of the cache and/or the WLAB cannot hold a data block long enough to keep up the locality burst. When both the local caches and the WLAB are large enough, the write traffic reduction becomes almost a constant and dependant solely on the characteristics of the trace references and the memory-cache speed dispatch. For a reference sequence tracing for I/O-oriented jobs, especially for the write oriented jobs, the write traffic reduction could be up to twenty or thirty percent. The second series of Figures 5.22-5.23 illustrates the impact of the memory-cache access time difference. The write traffic reduction is a direct proportion to the dispatch ratio of the memory-cache access time. This makes meaningful sense since a faster speed of the caches and a sufficiently large Write Look-Ahead Buffer keeps the writing data staying in the WLAB for a longer time relative to the cache access cycles. It greatly increases the opportunities for the succeeding memory write through operations to hit the same location of the WLAB as that of the previous write operation, before the data is actually written through. Note that Figures 5.19-5.20 show the performance of the WLAB cache systems based on a single, dual, and quadruple processor environments and 4:1, 2:1 and 1:1 in the parentheses of the legend refer to the memory-cache speed dispatch. We choose the dispatch ratio to make these series in as close as possible simulating circumstances. A large number of multiprocessor systems have a larger memory traffic and a smaller write traffic reduction for the same size of the caches and the WLAB. These results point out that a great degree of multiprocessors usually need a large size of the local caches and the Write Look-Ahead

Buffer to catch up the same performance.

Much work has been conducted to test the usefulness and utilization of the Write Look-Ahead Buffer. Figures 5.24-5.26 epitomize part of these simulation results from various configurations in different situations. Contrary to one's first thoughts, the smaller the size of the WLAB does not imply a higher percentage of utilization. Series 1 of Figure 5.24 displays the slowly increasing buffer utilization for the buffer size varying from 2, 4, 8, 16 and 32 blocks. Nonetheless, the utilization and performance of the buffer depends heavily on the number of multiprocessors and the dispatch ratio of main memory access time to the local cache access time. Figure 5.25 draws these influences of the dispatch ratio from 1:1, 2:1, 3:1 and 4:1 on a dual-processor system. The utilization increases but the performance might decrease when dispatch ratio is increased. Series 2 (buffer collision) and Series 3 (bus request failure) of Figures 5.24 through 5.26 reveal the performance of the Write Look-Ahead Buffer. Specifically, Series 3 represents the percentage of times when a local cache has to wait for bus that is currently busy,

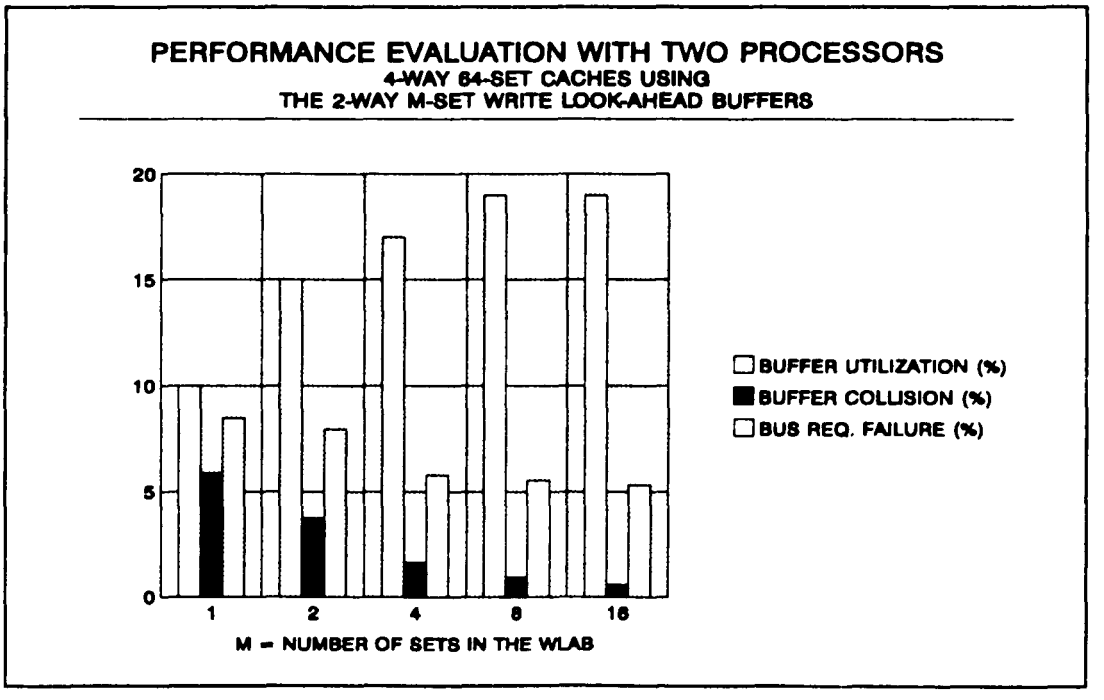

Figure 5.24 The Write Look-Ahead Buffer Utilization and Collision

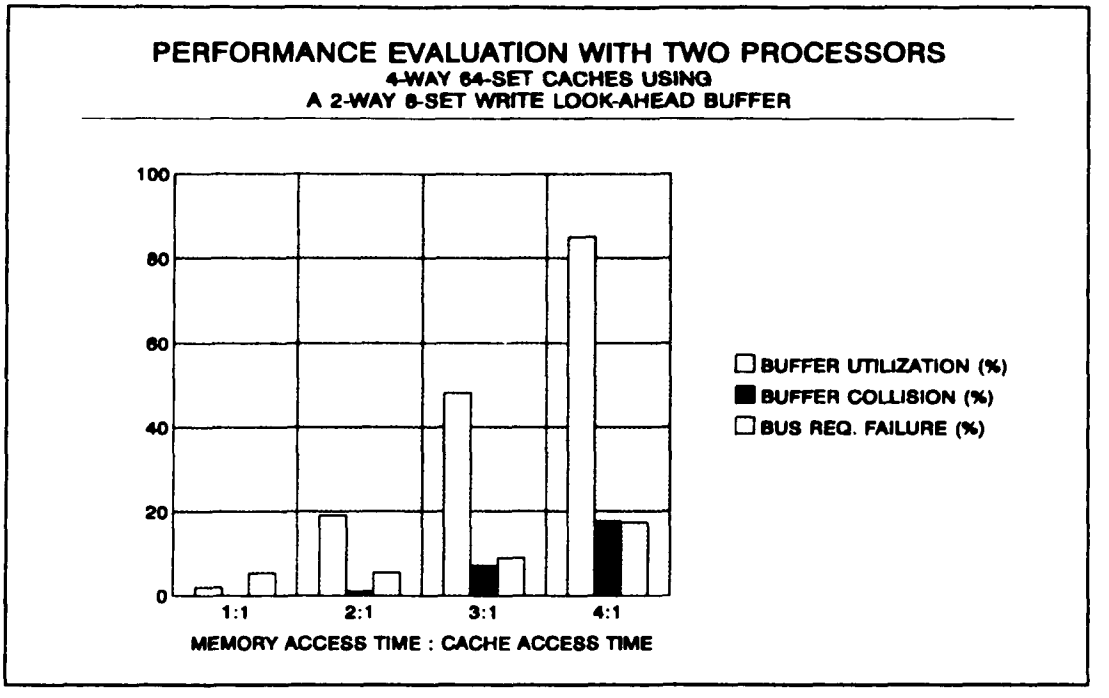

Figure 5.25 The Write Look-Ahead Buffer Utilization and Collision

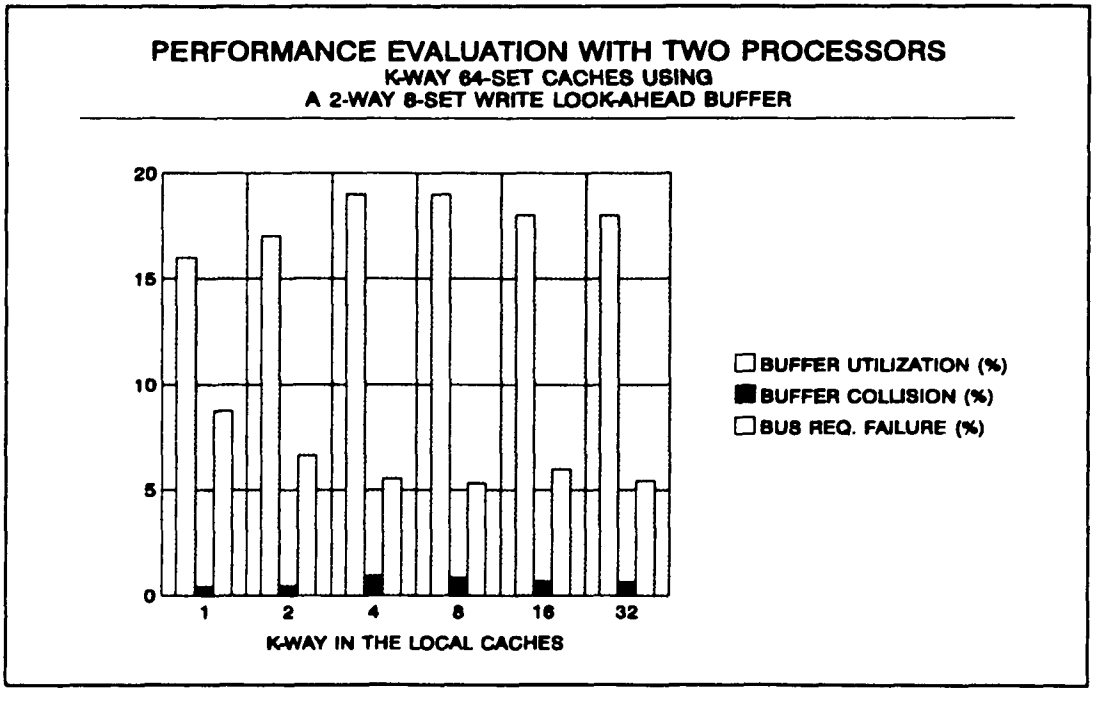

Figure 5.26 The Write Look-Ahead Buffer Utilization and Collision

while Series 2 describes the percentage of block collision, a situation when the block of the buffer requested by a local processor is not available because the previous data has not yet been written through. We expect both percentages to be as low as possible.

Many heavy and time-consuming trace simulations of the WLAB cache systems have been carried out - and are still being carried out - with specification in the PAISLey language. We expect that more valuable and significant consequences will be discovered in further optimum configuration analysis and research. However, our goal for this thesis, which we believe we have achieved, is to demonstrate how a concise and elegant operational specification paradigm using applicative high order logic can be applied to obtain useful design simulation results.

6. CONCLUSION AND FUTURE RESEARCH

This thesis has endeavored to present an operational approach to computer system specifications by exercising an applicative language, PAISLey, to provide truly implement-free simulation at the design stage. We asserted that a formal executable specification is not merely a precise and faithful representative of computer system design for an automatic consistency and correctness checking, but it also should be an operating model of the target object for an early, complete, technologically-independent simulation and should be capable of acquiring inherent performance measurements. By the execution of a specification, the performance measures for the specified design can be collected instantly from the running results as illustrated herein through various examples, most notably from the WLAB cache system. The executability of specification combines computer system description and simulation into one stage, which secures the original design against deceptive instantiation and artificial constraints raised by an improper pre-maturely selected implementation technique and/or the limitation produced during the conversion to a simulative version. Consequently, an early performance evaluation can be carried out more easily, accurately and faithfully through an executable specification, because of the operating paradigm established at the beginning. Tolerance of incomplete specifications makes PAISLey even more appropriate for the system level description which normally has many activities with unknown entities and requires more testability and flexibility.

We have used PAISLey to establish a computable paradigm of the Write Look-Ahead Buffer cache architecture for the multiprocessor systems. It has demonstrated how to validate system design as well as how to collect performance measurements. We have seen the powerful expressiveness of applicative high order logic. Nonetheless, computer hardware system specification using applicative high order logic is still in a primary stage, though a few applicative

languages have been adapted to fit this sphere. Many efforts have to be devoted to make practical. Therefore, one of our next plans is to apply the PAISLey specification to Very Long Instruction Word (VLIW) super-computer architectures [FisR91]. In order to validate and measure a VLIW structure, we have to specify many interactions between VLIW hardware and its optimized compiler. Most computer hardware description languages fail at - or at least have difficulty supporting - this circumstance because of their imperative style and "hardwired" or "circuitry"-oriented flavor. An applicative language like PAISLey, characterized by the power of abstraction and tolerance of incompleteness, has a tremendous capability to deal with such situations. We will see how well the PAISLey specification works with VLIW computer systems.

A WLAB cache system presents a cache coherence protocol that is designed initially to provide consistent use of the shared data blocks with reduced miss penalty and memory bus traffic for a tightly-coupled multiprocessor system using a single memory module. We would like to emphasize again that the Write Look-Ahead Buffer is nether a conventional cache, as a first level memory in a memory hierarchy, nor is it a recently-developed second level global cache in a cache hierarchy. The function of the look-ahead buffer is simply a write multi-buffer to speed up the write through and write miss operations by letting the actual writing take place in parallel to the local processing, via the internal bus between the WLAB buffer and the shared memory. The look-ahead structure of this write buffer efficiently and painlessly resolves the cache coherence problems. In addition, the look-ahead property also reduces the read miss latency by allowing quick reading the blocks waiting for writing through, or left in the buffer after being written back to the shared memory. As we know, the raising of hit ratio means the reduction of the shared memory accesses, i.e., the reduction of the memory bus traffic, which is the goal pursued by all the designers of the shared memory multiprocessor systems.

We exploited PAISLey to establish a computable paradigm to test, verify and evaluate its performance. We reap our first fruit satisfactorily with these initial results and by our extensive experiments through execution of the PAISLey specification. A variety aspects of this proposed

WLAB system have been measured and evaluated, especially: the effect of the Write Look-Ahead Buffer with respect to memory-bus traffic; the impact of the buffer to the reduction of miss rate and latency; the utilization of the Write Look-Ahead Buffer; the influence of speed dispatch between the Write Look-Ahead Buffer, the local caches and the main memory, and; the pertinence of the number of processors. We would like to continue the simulation and analysis of the WLAB cache systems, and attempt to find statistical optimization for the configuration parameters of the Write Look-Ahead Buffer regarding of the size and structure of local caches, the number of processors, the dispatching memory access time, and so on. Finding virtues of this in our originally-developed WLAB system is always exciting and enjoyable work.

So far, our trace simulation of the WLAB structure is traditionally based on word references. Nowadays, people start to build the computer systems around loosely organized multi-processors or local area network centered by one or two file servers, plus dozens of workstations. The best referencing unit to test performance in these circumstances would be packets rather than words. The critical problem here is how to produce a good packet sequence, which now is mostly derived by mathematically pseudo-methods. Though the derivation of an optimum packet sequence is absolutely beyond the range of this dissertation, we have interest in using one to promote further potential values of the WLAB cache hierarchical organization.

APPENDIX A: A Quick Survey of CHDLs

Before beginning, two points have to be cleared up. First of all, this list is not intended to exhaust all the CHDLs, an unrealistic task since hundreds of CHDLs have been invented and dozens more are added every year from all over the world. Instead, only those, on the basis of their characteristics as representatives of a class of CHDLs, are selected. Secondly, this part represents our earlier work to search for the right computer hardware description language to accomplish our missions, rather than to devise one. The work is was done during the end of 1989. We are sure many new languages have been developed since then. Only a minor revision was made and nearly no new languages, except those which have to be there because of their representation and "*typicalness*" for the current research and future tendency, have been added to this list. The major source of remarks comes from ([Ayl86, Barb75, BarbU85, BergZ86, BorW91, Chu74, DarR89, Das84a, 89b, Hart87a, 87b, NasS86, Su77, ZavS86]). As attested to by the following list, not much work has been done on the description of parallel processing and multiprocessor systems and architectures. The order of the list attempts to follow the historical time when the languages were developed and put into use.

CDL ([Chu72a, 72b, Mes82]) - An Argol-like register transfer language (TRL), that has been implemented in FORTRAN for simulation and design automation purposes. CDL, conceived in 1965, is an earlier excellent representative of the class of nonprocedural languages, i.e., languages which attach no meaning to the lexicographical ordering of statements describing the operation of the system. CDL allows the designer to describe parallel, as well as sequential, operations. It is neither hierarchical nor modular. CDL provides both structural and functional digital system descriptions. Design verification

is achieved through simulation.

DDL ([DulD68, Dul70, MarF85, UehS83) - One of the earliest TRLs, DDL is still used as it provides the descriptive basis for a formal hardware verification system developed at the Fujitsu laboratories in Japan.

APDL ([Dar69]) - A register transfer language that represents the class of Argol-like TRLs; that is, languages based on Argol with extensions to handle timing and register variables.

APL ([Ive62a, 62b]) - An RTL that provides the richness of the set of data operators and the facility to handle arrays. A major drawback is the lack of facilities to describe parallel activities. It is used in the Alert System developed at IBM as a front end for the design and logic automation systems.

ISP ([Bel71, Chu72a]) - Developed initially to describe the primitives of the programming level of design, it handles concurrency, and sequencing of activities in a simple fashion and provides an adequate set of data and control operators.

TEGAS ([Szy72]) - A general-purpose digital test generation and simulation system that furnishes functional simulation, fault simulation and test generation. The TEGAS system has its own design description language called TDL. The TDL description is translated by a preprocessor in the TEGAS system into internal forms, that are used by simulators.

AHPL ([HilP78]) - A hardware programming language based on the notational conventions of APL. AHPL is procedural, i.e., the order of statements is significant. It is oriented more to the register transfer level. AHPL was conceived, implemented, and continues to be improved and used at the University of Arizona.

ZEUS ([LieK83]) - A general-purpose hardware description language for VLSI, which provides facilities for describing circuits by recursive and interactive methods. These methods permit changing a few elements in the description and recompiling it for specification and functional verification of parameterized families of design. ZEUS can provide both structural and functional descriptions. Compared to the other HDLs, ZEUS extends the hardware structuring facilities, especially type assignments. ZEUS adds constraints to a circuit's layout.

BDSYN ([Seg89]) - A specification language that describes and implements combinational logic only. It takes as input a textual description of a block of combinational logic and produces a collection of logic functions which implement the described functions. BDSYN was written to be a front end of the Berkeley synthesis system.

STRUM ([Pat76, 81]) - A machine-dependent microprogramming specification. It uses structured programming techniques for correcting firmware and verification.

S* ([Das84b]) - A microprogramming language schema that consists of two completely defined members: S*A, a general purpose procedural language for the formal, operational specification of exo-architectures and endo-architectures, and; S*, a high-level microprogramming language schema. S* is instantiated at Nanodata QM-1, oriented toward a user-microprogrammable computer.

AADL ([Dam85, DamD86]) - An architecture description language that is designed for describing micro-architectures and exo-architectures. Essentially, the language is used to create and verify the microprogramming languages form the language schema S*. It can specify a poly-phase execution sequence with overlapping phases. The heart of AADL lies in micro-operations and exo-architectural level.

MIDL ([Sin81]) - A micro-architecture description language that views micro-machines in four parts: data store declarations, schedules, operations and operands. It supports a variety of data types and makes the distinction between static and transient stores. It is not entirely clear from the original how the schedules for poly-phase timed architecture can be defined.

ISPS ([Barb81, Wilk82]) - Instruction Set Processor Specification, that is the second implementation of ISP as a computer specification language. It is towards the formalization of the digital design process at higher or behavioral levels with limited capabilities for describing the gate level. Besides simulation and synthesis of hardware, it provides software generation, program verification and hardware architecture evolution and control. ISPS can be considered as a high-level programming language, though its notation is developed for digital systems. However, it is not suitable for micro-architecture specification. ISPS project was supported by DOD.

HSL ([Pia84]) - A language proposed as being capable of describing hardware ranging from the circuit level up to both exo-architecture and PMS (system) levels of abstraction. Thus, it provides both structural and behavioral description. It also shows the possibility of inter statement parallelism and parallel execution of modules. HSL is a strongly typed language patterned after Ada.

CONLAN ([Pil80a, 80b]) - A consensus hardware description language developed by an international group. The objectives of the CONLAN project are toward the standardization of the existing HDLs onto a common language. In the CONLAN approach, a family of different hardware description languages can be derived for special applications, all sharing a common semantic base - a base consensus language. It is a general, formal language construction mechanism for description of hardware and firmware at different levels of abstraction. But, CONLAN project has difficulty in

establishing the system tools, such as a simulator or analyzer, because of the consolidation of a number of application oriented HDLs.

IDL ([MaiO82, MaiO84, MaiP83]) - An interactive design language whose initial purpose was to facilitate the design of PLAs embodying highly complex algorithms. It is now used for random logic design as well. IDL is nonprocedural. Data entry may be in the form of a graphical flow chart or in text form. IDL is implemented in ADL.

TI-HDL ([Ayl86]) - Texas Instruments Hardware Description Language that is a block-structured hardware design and description language. TI-HDL is capable of multi-level description, i.e., it can outline a digital design from abstract functional behavior down through detailed logical design.

S*M ([WilsD89]) - A language designed to provide functional descriptions of the behavioral characteristics of a micro-architecture. In particular, S*M is intended for describing clocked micro-architecture, i.e., micro-architecture whose operation and behavior are under the control of one or more synchronized clock. S*M was developed from a revised model of clocked micro-architectures originally proposed by Dasgupta ([Das84b]).

VHDL ([ShaL85, LipsM86, IEE87]) - A multi-level hardware description language whose development was sponsored by DOD to specify VHSIC. VHDL supports the design, documentation, and efficient simulation of hardware from the digital system level to gate level. VHDL is capable of both structural and behavioral description. Behavior is specified operationally, and the flow of control through such descriptions may be both procedural and/or nonprocedural. Like HSL, VHDL adopts many concepts from Ada. VHDL was intended to become the standard CHDL in all DOD hardware design projects and is also accepted by IEEE as the CHDL standard. Since VHDL is a quite new

language (at the time when this search was carrying), there are several dimensions across which the language needs to be empirically evaluated, e.g., its ability to incorporate design tools into a given design environment, and the extent to which it facilitates formal design verification. The IEEE 92 version of VHDL, being developed recently, may have answer to these questions.

ACE ([Wils87, 89]) - A very new hardware description language that is intended for the behavioral description of machines at several abstraction levels. Semantics of ACE are defined formally using a temporal logic, this definition establishes the foundation for the formal verification of machine design. Since ACE is very new, it needs to demonstrate its effectiveness and descriptive capacities.

EXEL ([DutG89]) - A language for behavioral synthesis. EXEL illustrates the use of modern user interface techniques to enhance ease-of-use. It is built in SUN workstations and takes advantages of SUNVIEWS, a user interface building tool kit. EXEL users interact with the design system by manipulating flowgraphs and text on a screen with back-end tools, capturing additional behavioral and structural information implied by the icons and their connections. It is also indicated that EXEL design representation can be translated into VHDL input text and can therefore be used as input to VHDL synthesis systems. However, its verification ability remains to be demonstrated.

ODICE ([MulR89]) - A hardware description in a CAD environment that incorporates object-oriented features. Object-Oriented Design (OOD) is a powerful method to build abstract data types and to encapsulate and hide information. ODICE maps classes onto hardware component types and class inherence onto hierarchies of components. The methods or object manipulation procedures represent the behavior of the hardware components. ODICE does not support multiple inherence in the OOD sense.

STATEMATE ([Hare87, HareL88]) - A graphic working environment, intended for the specification, analysis, design and documentation of large and complex reactive systems, such as real time embedded systems, control and communication systems, and interactive software. It enables a user to prepare, analyze and debug diagrammatic, yet precise descriptions of the system under development from three inter-related points of view - capturing structure, functionality and behavior. Its further attempts are to be executable specification and visual formalism.

PAISLey ([Zav82a, 83]) - An executable specification language that is especially well suited to real-time and distributed systems. It is motivated by an approach to software development based on the separation of problem-oriented from implementation-oriented concerns, and promising several substantial benefits over conventional development cycles, including maximum parallelism, encapsulated computation, toleration of incompleteness, consistency checking and coherence. The language is executed by an interpreter that provides capabilities for debugging specifications, giving demonstrations to customers early performance simulation, and eventually rapid prototyping. A recent attempt of PAISLey is trying to specify the computer hardware architectures, especially in parallel and multi-processor systems. PAISLey is developed in AT&T Bell Laboratories, running under the UNIX system and following the conventions of the UNIX program interface.

PROT-NET ([BruM86]) - PROT nets are extended Petri Nets. A PROT-net based model supports the process approach to requirement specification and is suitable for carrying out system simulation. It lends itself to the rapid prototyping of a system since PROT nets are shown to be translatable into Ada program structures. Thus, PROT-net is an operational approach to system specification, e.g., a flexible manufacturing system specification.

Interval Temporal Logic (ITL) ([Boc82, FujF89, Hal87, Mos84, 85, NarA88, Wils89]) - The earliest work of computer specification with temporal logic was by Bochmann ([Boc82]) and is concerned with asynchronous-timed systems. The second work with temporal logic for machine description was developed by Moszkowski ([Mos84, 85]), which deals with circuit level description. Recently, Wilsey ([Wils89]) adopted interval temporal logic to describe the timing scheme or clocked micro-architecture, and he also formalized the semantics of ACE, a new multi-level CHDL, with temporal logic. M. Fujita and H. Fujisawa use propositional temporal logic for specification, verification and synthesis of control circuits ([FujF89]). Temporal logic has been advocated as a suitable formalism for behavioral description. In the framework of its logic, it is possible to specify qualitative and quantitative aspects of system. To provide for a greater degree of abstractness in specifications, temporal interval logic has been advocated as a formalism for reasoning about concurrent systems. Due to its strict logic formalism, an accurate interpretation of hardware behavior can be provided, and the correctness of the description can be easily and formally verified. That is why temporal logic is now getting more and more attractive in the synthesis and verification of computer hardware design. The major drawback of temporal logic is that there exist a variety of different notations and variants, making it hard to read and understand temporal logic specification. It also lacks implementation. Specification with temporal logic for parallel or multiprocessor computers needs further exploration since temporal logic is adequate for describing the behavior of concurrent systems. Incidentally, a lot of work on temporal logic has been done with real-time systems.

Tempura ([Hal87]) - An executable subset of Interval Temporal Logic (ITL). It only includes the logic constructs that would make a formula possess one model for execution efficiency. A Tempura interpreter evaluates formulas by recursively transferring them into sequences of temporal states. It verifies the specification through simulation.

FCP ([DotA90]) - An abbreviation of Flat Concurrent Prolog which is an applicative logic programming language. It belongs to the Prolog family. FCP programs comprise a collection of finite guarded clauses which define relations among entities. Hardware components are expressed by entities and are connected by an AND operator in the clause and the shared variables that function as communication channels. Standard simulation or symbolic simulation can be performed for automatic check of design correctness. However, for symbolic simulation, additional tools are needed to process boolean algebra.

TRIO ([CoeM91, GheM90]) - An extended temporal logic language for executable specification of real time systems. It uses first order logic augmented with temporal operators to construct computer description. It is different from classical temporal logic in that it introduces a quantitative expression of time which allows the measurement of time intervals. The salient feature of TRIO is that it supports several forms of hardware verification.

APPENDIX B: SPECIFICATION OF A WLAB CACHE SYSTEM FOR 4-MULTIPROCESSORS

```
/*  A PAISLey specification of a single cache system using a write look-ahead buffer  */

#include "multi.h"*

"Configuration of A K-Way Set-Associative Cache"

    CACHE = #1.. Nset < *  SET > ;
    SET = #1.. Kway < *  (VALID*TAG*LINE) > ;
    LINE = INTEGER | FILLER ;
    VALID = { 0, 1 } ;

    ADDRESS = TAG*SETNO*WORDNO ;
    TAG = INTEGER ;
    SETNO = { J#1.. Nset  < , J> } ;
    LINENO = { J#1.. Kway  < , J> } ;
    WORDNO = { J#1.. Nword < , J> } ;

"Configuration of A Write Look-Ahead Buffer"

    WLA-BUFFER = ADDR-QUEUE*BUFFER ;
    ADDR-QUEUE = #1.. Qlen < *  (B-SETNO*B-LINENO) > ;
    BUFFER = #1.. Bset < *  B-SET > ;
    B-SET = #1.. Bway < *  (DIRTY-CNT*TAG*LINE) > ;
    B-SETNO = { J#1.. Bset  < , J> } ;
    B-LINENO = { J#1.. Bway  < , J> } ;
```

* Both header file "multi.h" and "mnext.h" store the parameters relevant to the size of cache, the WLAB, and the reference trace files, etc. By virtue of the UNIX "*make*" utility, these header files ease the tune of various system configurations.

```
    DIRTY-CNT = { J#0.. Nword < , J> } ;

"Configuration for Performance Parameters"

    REFERENCE = ADDRESS*RW-REFER ;
    RW-REFER = {R, W, I} ;
    BUF-REFER = {Rmis, Wmis, Whit} ;

    HIT-STATUS = { Hit, Fail } ;
    LOCK-STATUS = { 0, 1 } ;

System Declaration - Cold Starting Trace Simulation"

    ( memory-bus-cycle[initial-bus] ,
      global-lock-cycle[(0, 0)] ,
      p#1.. Ncpu
        < , cache-p-cycle[(0, initial-cache-p)] >
    ) ;

"Initialization"

    initial-bus: --> INTEGER ;
    initial-bus = proj[(1, (0, initial-buffer))] ;

    initial-buffer: --> FILLER ;
    "initial-buffer = simulation provided procedure (multi.c)"

    p#1.. Ncpu
        < ;   initial-cache-p: --> CACHE ;
              initial-cache-p = ( J#1.. Nset < , (K#1.. Kway < , (0,-1,Null) >) >)
        > ;

"Memory Bus Cycle - Process for Writing Dirty Words Back to Memory"

    memory-bus-cycle: INTEGER --> INTEGER ;
    memory-bus-cycle[count] = proj[(1, (sum[(count, 1)], memory-write-phase))] ;
```

```
    memory-write-phase: --> FILLER ;
    memory-write-phase = /   equal[(is-buf-empty, 0)]: Null ,
                             True: write-word-to-memory
                         / ;

    is-buf-empty: --> INTEGER ;
    "is-buf-empty = simulation provided procesure (multi.c)"

    write-word-to-memory: --> FILLER ;
    "write-word-to-memory = simulation provided procedure (multi.c)"

"Bus Synchronization Cycle"

    global-lock-cycle: INTEGER*LOCK-STATUS --> INTEGER*LOCK-STATUS ;
    global-lock-cycle[(count, signal)] = (sum[(count, 1)], send-signal[signal]) ;

    send-signal: LOCK-STATUS --> LOCK-STATUS ;
    send-signal[signal] = xr-signal[signal] ;

    p#1.. Ncpu
        < ;   get-signal-p: { 1 } --> LOCK-STATUS ;
              get-signal-p[signal] = xm-signal[signal] ;

              set-signal-p: { 0 } --> LOCK-STATUS ;
              set-signal-p[signal] = xm-signal[signal] ;

              get-signal-p: ! ub 5.0 ns ;
              set-signal-p: ! lb 20.0 ns , ub 30.0 ns
        > ;

    send-signal:  ! lb 1.0 ns ;

"Time Constraints - Cache speed versus memory bus speed"

    memory-bus-cycle: ! lb 20.0 ns, ub 20.0 ns ;
    p#1.. Ncpu < ; cache-p-cycle: ! lb 40.0 ns, ub 40.0 ns > ;
```

```
"Cache Reference Cycle"

    p#1.. Ncpu
        < ;
            cache-p-cycle: INTEGER*CACHE --> INTEGER*CACHE ;
            cache-p-cycle[(count-p, cache-p)] =
                ( sum[(count-p,1)], addr-map-p[(next-reference-p[count-p], cache-p)] ) ;

        "Fetching next reference"
            next-reference-p: INTEGER --> REFERENCE ;
            "next-reference-p[count] = Developed PAISLey I/O utilities"

            addr-map-p: REFERENCE*CACHE --> CACHE ;
            addr-map-p[(((tag, set-no, wd-no), r-w), cache)] =
                / equal[(associative-map-p[(tag, proj[(set-no,cache)], r-w)], (r-w, Hit))]:
                        hit-processing-p[(((tag,set-no,wd-no), r-w), cache)],
                  True:  miss-processing-p[(((tag,set-no,wd-no), r-w), cache)]
                / ;

            hit-processing-p: REFERENCE*CACHE --> CACHE ;
            hit-processing-p[(((tag, set-no, wd-no), r-w), cache)] =
                / or[(equal[(r-w, I)], equal[(r-w, R)])]: cache,
                  equal[(get-signal-p[1], 0)]:
                        proj[(1, (write-through-p[(      (tag, set-no, wd-no), cache,
                                get-buf-set[set-mapping[(tag,set-no)]] )],
                                set-signal-p[0]) )],
                  True: proj[(1, (cache, wait-signal-p))]
                / ;

            write-through-p: ADDRESS*CACHE*B-SET --> CACHE ;
            write-through-p[((tag, set-no, wd-no), cache, buf-set)] =
                / equal[ (buf-map-p[(tag-mapping[(tag,set-no)], buf-set, Whit)],
                                (Whit, Hit)) ]:
                        proj[(1, (cache, write-to-buffer[(tag-mapping[(tag,set-no)],
                                        set-mapping[(tag,set-no)], wd-no)]) )],
                  is-buf-available-p[(buf-set)]:
                        proj[(1, (cache, write-to-buffer[(tag-mapping[(tag,set-no)],
```

```
                                set-mapping[(tag,set-no)], wd-no)]) )],
          True: proj[(1, (cache, wait-on-line-p))]
                                "because of the wlab unavailable"
        / ;

miss-processing-p: REFERENCE*CACHE --> CACHE ;
miss-processing-p[(ref, cache)] =
        / equal[(get-signal-p[1], 1)]:  proj[(1, (cache, wait-signal-p))],
          True: rw-miss-processing-p[(ref, cache)]
        / ;

rw-miss-processing-p: REFERENCE*CACHE --> CACHE ;
rw-miss-processing-p[(((tag, set-no, wd-no), r-w), cache)] =
        / or[(equal[(r-w, I)], equal[(r-w, R)])]:
                read-miss-p[(  (tag, set-no, wd-no), cache,
                                get-buf-set[set-mapping[(tag,set-no)]] )],
          True: write-miss-p[( (tag, set-no, wd-no), cache,
                                get-buf-set[set-mapping[(tag,set-no)]] )]
        / ;

read-miss-p: ADDRESS*CACHE*B-SET --> CACHE ;
read-miss-p[((tag, set-no, wd-no), cache, buf-set)] =
        / equal[( buf-map-p[(tag-mapping[(tag,set-no)], buf-set, Rmis)],
                        (Rmis, Hit) )]
                : proj[(1, (update-cache-p[((tag,set-no,wd-no), cache)],
                                        set-signal-p[0]) )],
          True  : proj[(1, (update-cache-p[((tag,set-no,wd-no), cache)],
                                        set-signal-p[0]) )]
        / ;

write-miss-p: ADDRESS*CACHE*B-SET --> CACHE ;
write-miss-p[((tag, set-no, wd-no), cache, buf-set)] =
        / equal[( buf-map-p[(tag-mapping[(tag,set-no)], buf-set, Wmis)],
                        (Wmis, Hit) )]:
                proj[(1, (update-cache-p[((tag, set-no, wd-no),cache)],
                        write-to-buffer[ (tag-mapping[(tag,set-no)],
                                        set-mapping[(tag,set-no)], wd-no)],
```

```
                    set-signal-p[(0])  )],
        is-buf-available-p[(buf-set)]:
              proj[(1, (update-cache-p[((tag, set-no, wd-no),cache)],
                    write-to-buffer[(tag-mapping[(tag,set-no)],
                                          set-mapping[(tag,set-no)], wd-no)],
                    set-signal-p[(0])  )],
        True: proj[(1, (cache, wait-on-line-p))]
                          "because of the wlab unavailable"
      / ;

associative-map-p: TAG*SET*RW-REFER --> RW-REFER*HIT-STATUS ;
associative-map-p[(key, (j#1.. Kway <, (valid-j,tag-j,line-j) >), r-w)] =
      / j#1.. Kway < , and[(equal[(valid-j,1)], equal[(key,tag-j)])]: (r-w, Hit) >,
        True: (r-w, Fail)
      / ;

buf-map-p: TAG*B-SET*BUF-REFER --> BUF-REFER*HIT-STATUS ;
buf-map-p[(tag-key, (j#1.. Bway < , (d-cnt-j, tag-j, line-j) >), r-w)] =
      / j#1.. Bway <,  equal[(tag-key, tag-j)]: (r-w, Hit) >,
        True: (r-w, Fail)
      / ;

is-buf-available-p: B-SET --> BOOLEAN ;
is-buf-available-p[(j#1.. Bway < , (d-cnt-j, tag-j, line-j) >)] =
      / j#1.. Bway <, equal[(d-cnt-j, 0)] : True >,
        True: False
      / ;

update-cache-p: ADDRESS*CACHE --> CACHE;
update-cache-p[((tag, set-no, wd-no), cache)] =
      replace[(load-cache-line-p[(tag, proj[(set-no,cache)])], set-no, cache)] ;

load-cache-line-p: TAG*SET --> SET ;
load-cache-line-p[(tag, (j#1.. Kway <, (valid-j,tag-j,line-j)>))] =
      / equal[(Kway, 1)]: (1, tag, Null),
        k#1.. Kway < , equal[(valid-k,0)]:
                    replace[((1, tag, Null), k,
```

```
                                        (j#1.. Kway <, (valid-j,tag-j,line-j)>) )]
                                >,
                    True: replace[( (1, tag, Null), select-victim-p,
                                        (j#1.. Kway < , (valid-j, tag-j, line-j) >) )]
                /;

        select-victim-p: --> LINENO ;
        "Value of select-victim is generated arbitrarily by setmode random select-victim-p"

        wait-signal-p: --> INTEGER ;
        "wait-signal-p = simulation provided procedure (mnext.c)"
        "it returns a number of wasted waiting cycles of processor p"

        wait-on-line-p: --> INTEGER
        "wait-on-line-p = simulation provided procedure (mnext.c)"
        "it returns a number representing the total waiting cycles since start"

    > ;

get-buf-set: B-SETNO --> B-SET ;
"get-bus-set[b-set-no] = simulation provided procedure (multi.c)"

write-to-buffer: TAG*B-SETNO*WORDNO --> FILLER ;
"write-to-buffer[(tag, b-set-no, wd-no)= simulation provided procedure (multi.c)"

"Address mapping between the write look-ahead buffer and the locacl caches"

tag-mapping: TAG*SETNO --> TAG ;
tag-mapping[(tag, set-no)] = quot[(sum[(prod[(tag, Nset)], set-no)], Bset)] ;
                                        "buf_tag = (tag*Nset+set_no) div Bset"

set-mapping: TAG*SETNO --> B-SETNO ;
set-mapping[(tag, set-no)] = mod[(sum[(prod[(tag, Nset)], set-no)], Bset)] ;
                                        "buf_set_no= (tag*Nset+set_no) mod Bset"
```

```
/*  C utilities developed to enhance the batch input mode of the PAISLey language. Here, we list
    only those functions especially for reading a sequence of trace references as follows:

        next-reference-p: INTEGER --> REFERENCE ;       get a next reference
        wait-on-line-p: --> INTEGER ;          waiting because wla-buffer is unavailable
        wait-signal-p: --> INTEGER ;           waiting because bus is unavailable

    where p=1,2, .., Ncpu
*/


#include </usr/paisley/include/pdefines.h>
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include "mnext.h"*
#define LineLen  80

P_Valueptr      P_makevalue(), P_maketuple();

static  long    WaitBuf[Ncpu], WaitSig[Ncpu], NumAddr[Ncpu], idx[Ncpu],
                tag_lst[Ncpu][MaxNo],setno_lst[Ncpu][MaxNo],wdno_lst[Ncpu][MaxNo];
static  char    *rw_lst[Ncpu][MaxNo];
static  FILE    *fp[Ncpu][Mdeg];


/* For the first processor */

P_Valueptr next_reference_1 ( addr_cnt )
 P_Valueptr addr_cnt;
 {
   int          P=0, cnt, tag1, setno1, wdno1, InCnt;
```

* Both header file "multi.h" and "mnext.h" store the parameters relevant to the size of cache, the WLAB, and the reference trace files, etc. By virtue of the UNIX "*make*" utility, these header files ease the tune of various system configurations.

```
char        *rw1;
P_Valueptr p_tag, p_setno, p_wdno, p_rw, p_addr, q;
P_Atom     tag, setno, wdno, rw;

if ((cnt=addr_cnt->cvalue.integer)==0)
{
 WaitBuf[P]=WaitSig[P]=NumAddr[P]=0;
 for (InCnt=0; InCnt<Mdeg; InCnt++)
      if ( (fp[P][InCnt]=fopen(fname[P][InCnt], "r"))==NULL )
      {
       fprintf(stderr, "Can't open the input trace file %s !/n", fname[P][InCnt]);
       exit(1);
      }
 for (InCnt=0; InCnt<MaxNo; InCnt++)
      rw_lst[P][InCnt]=calloc(2, sizeof(char));
 rw1=calloc(2,sizeof(char));
}
idx[P]=(cnt-WaitBuf[P]-WaitSig[P])%MaxNo;
if (idx[P]==0 && Preidx[P]!=idx[P])
{                              /* Avoid 2nd time read if waiting occurs */
 InCnt=0;
 while ((InCnt<MaxNo) &&
        (fscanf(fp[P][mdix[P]], "%d%d%d%s", &tag1, &setno1, &wdno1, rw1)==4))
 {
      tag_lst[P][InCnt]=tag1;
      setno_lst[P][InCnt]=setno1+1;
      wdno_lst[P][InCnt]=wdno1+1;
      strcpy(rw_lst[P][InCnt++], rw1);
 }
 NumAddr[P]=NumAddr[P]+InCnt;
 mdix[P]=(mdix[P]+1)%Mdeg;
}
Preidx[P]=idx[P];
tag.integer=tag_lst[P][idx[P]];
setno.integer=setno_lst[P][idx[P]];
wdno.integer=wdno_lst[P][idx[P]];
rw.symbol=rw_lst[P][idx[P]];
```

```
    p_tag=P_makevalue(INTEGER, tag);
    p_setno=P_makevalue(INTEGER, setno);
    p_wdno=P_makevalue(INTEGER, wdno);
    p_rw=P_makevalue(SYMBOL, rw);
    p_addr=P_maketuple(p_tag, p_setno, p_wdno, 0);
    q=P_maketuple(p_addr, p_rw, 0);
    return q ;
  }

P_Valueptr wait_on_line_1 ()
  {
    int  P=0;          P_Atom  a_wait;       P_Valueptr  p_wait;

    a_wait.integer= ++(WaitBuf[P]) ;
    p_wait=P_makevalue(INTEGER, a_wait) ;
    return p_wait ;
  }

P_Valueptr wait_signal_1 ()
  {
    int  P=0;          P_Atom  a_wait;       P_Valueptr  p_wait;

    a_wait.integer= ++(WaitSig[P]) ;
    p_wait=P_makevalue(INTEGER, a_wait) ;
    return p_wait ;
  }

/* For the second processor */

P_Valueptr next_reference_2 ( addr_cnt )
  P_Valueptr addr_cnt;
  {
    int          P=1, cnt, tag1, setno1, wdno1, InCnt;
    char         *rw1;
    P_Valueptr p_tag, p_setno, p_wdno, p_rw, p_addr, q;
    P_Atom     tag, setno, wdno, rw;
```

```
if ((cnt=addr_cnt->cvalue.integer)==0)
{
 WaitBuf[P]=WaitSig[P]=NumAddr[P]=0;
 for (InCnt=0; InCnt<Mdeg; InCnt++)
      if ( (fp[P][InCnt]=fopen(fname[P][InCnt], "r"))==NULL )
      {
       fprintf(stderr, "Can't open the input trace file %s !/n", fname[P][InCnt]);
       exit(1);
      }
 for (InCnt=0; InCnt<MaxNo; InCnt++)
      rw_lst[P][InCnt]=calloc(2, sizeof(char));
 rw1=calloc(2,sizeof(char));
}
idx[P]=(cnt-WaitBuf[P]-WaitSig[P])%MaxNo;
if (idx[P]==0 && Preidx[P]!=idx[P])
{                              /* Avoid 2nd time read if waiting occurs */
 InCnt=0;
 while ((InCnt<MaxNo) &&
        (fscanf(fp[P][mdix[P]], "%d%d%d%s", &tag1, &setno1, &wdno1, rw1)==4))
 {
      tag_lst[P][InCnt]=tag1;
      setno_lst[P][InCnt]=setno1+1;
      wdno_lst[P][InCnt]=wdno1+1;
      strcpy(rw_lst[P][InCnt++], rw1);
 }
 NumAddr[P]=NumAddr[P]+InCnt;
 mdix[P]=(mdix[P]+1)%Mdeg;
}
Preidx[P]=idx[P];
tag.integer=tag_lst[P][idx[P]];
setno.integer=setno_lst[P][idx[P]];
wdno.integer=wdno_lst[P][idx[P]];
rw.symbol=rw_lst[P][idx[P]];
p_tag=P_makevalue(INTEGER, tag);
p_setno=P_makevalue(INTEGER, setno);
p_wdno=P_makevalue(INTEGER, wdno);
p_rw=P_makevalue(SYMBOL, rw);
```

```
    p_addr=P_maketuple(p_tag, p_setno, p_wdno, 0);
    q=P_maketuple(p_addr, p_rw, 0);
    return q ;
  }

P_Valueptr wait_on_line_2 ()
  {
    int  P=1;          P_Atom  a_wait;       P_Valueptr  p_wait;

    a_wait.integer= ++(WaitBuf[P]) ;
    p_wait=P_makevalue(INTEGER, a_wait) ;
    return p_wait ;
  }

P_Valueptr wait_signal_2 ()
  {
    int  P=1;          P_Atom  a_wait;       P_Valueptr  p_wait;

    a_wait.integer= ++(WaitSig[P]) ;
    p_wait=P_makevalue(INTEGER, a_wait) ;
    return p_wait ;
  }

/* For the third processor */

P_Valueptr next_reference_3 ( addr_cnt )
  P_Valueptr addr_cnt;
  {
    int          P=2, cnt, tag1, setno1, wdno1, InCnt;
    char         *rw1;
    P_Valueptr p_tag, p_setno, p_wdno, p_rw, p_addr, q;
    P_Atom     tag, setno, wdno, rw;

    if ((cnt=addr_cnt->cvalue.integer)==0)
    {
     WaitBuf[P]=WaitSig[P]=NumAddr[P]=0;
     for (InCnt=0; InCnt<Mdeg; InCnt++)
```

```
        if ( (fp[P][InCnt]=fopen(fname[P][InCnt], "r"))==NULL )
        {
         fprintf(stderr, "Can't open the input trace file %s !/n", fname[P][InCnt]);
         exit(1);
        }
   for (InCnt=0; InCnt<MaxNo; InCnt++)
        rw_lst[P][InCnt]=calloc(2, sizeof(char));
   rw1=calloc(2,sizeof(char));
  }
  idx[P]=(cnt-WaitBuf[P]-WaitSig[P])%MaxNo;
  if (idx[P]==0 && Preidx[P]!=idx[P])
  {                              /* Avoid 2nd time read if waiting occurs */
   InCnt=0;
   while ((InCnt<MaxNo) &&
          (fscanf(fp[P][mdix[P]], "%d%d%d%s", &tag1, &setno1, &wdno1, rw1)==4))
   {
        tag_lst[P][InCnt]=tag1;
        setno_lst[P][InCnt]=setno1+1;
        wdno_lst[P][InCnt]=wdno1+1;
        strcpy(rw_lst[P][InCnt++], rw1);
   }
   NumAddr[P]=NumAddr[P]+InCnt;
   mdix[P]=(mdix[P]+1)%Mdeg;
  }
  Preidx[P]=idx[P];
  tag.integer=tag_lst[P][idx[P]];
  setno.integer=setno_lst[P][idx[P]];
  wdno.integer=wdno_lst[P][idx[P]];
  rw.symbol=rw_lst[P][idx[P]];
  p_tag=P_makevalue(INTEGER, tag);
  p_setno=P_makevalue(INTEGER, setno);
  p_wdno=P_makevalue(INTEGER, wdno);
  p_rw=P_makevalue(SYMBOL, rw);
  p_addr=P_maketuple(p_tag, p_setno, p_wdno, 0);
  q=P_maketuple(p_addr, p_rw, 0);
  return q ;
}
```

```
P_Valueptr wait_on_line_3 ()
 {
   int  P=2;          P_Atom  a_wait;       P_Valueptr  p_wait;

   a_wait.integer= ++(WaitBuf[P]) ;
   p_wait=P_makevalue(INTEGER, a_wait) ;
   return p_wait ;
 }

P_Valueptr wait_signal_3 ()
 {
   int  P=2;          P_Atom  a_wait;       P_Valueptr  p_wait;

   a_wait.integer= ++(WaitSig[P]) ;
   p_wait=P_makevalue(INTEGER, a_wait) ;
   return p_wait ;
 }

/* For the fourth processor */

P_Valueptr next_reference_4 ( addr_cnt )
 P_Valueptr addr_cnt;
 {
   int          P=3, cnt, tag1, setno1, wdno1, InCnt;
   char         *rw1;
   P_Valueptr p_tag, p_setno, p_wdno, p_rw, p_addr, q;
   P_Atom     tag, setno, wdno, rw;

   if ((cnt=addr_cnt->cvalue.integer)==0)
   {
    WaitBuf[P]=WaitSig[P]=NumAddr[P]=0;
    for (InCnt=0; InCnt<Mdeg; InCnt++)
        if ( (fp[P][InCnt]=fopen(fname[P][InCnt], "r"))==NULL )
        {
         fprintf(stderr, "Can't open the input trace file %s !/n", fname[P][InCnt]);
         exit(1);
        }
```

```
    for (InCnt=0; InCnt<MaxNo; InCnt++)
        rw_lst[P][InCnt]=calloc(2, sizeof(char));
    rw1=calloc(2,sizeof(char));
   }
   idx[P]=(cnt-WaitBuf[P]-WaitSig[P])%MaxNo;
   if (idx[P]==0 && Preidx[P]!=idx[P])
   {                              /* Avoid 2nd time read if waiting occurs */
    InCnt=0;
    while ((InCnt<MaxNo) &&
           (fscanf(fp[P][mdix[P]], "%d%d%d%s", &tag1, &setno1, &wdno1, rw1)==4))
    {
        tag_lst[P][InCnt]=tag1;
        setno_lst[P][InCnt]=setno1+1;
        wdno_lst[P][InCnt]=wdno1+1;
        strcpy(rw_lst[P][InCnt++], rw1);
    }
    NumAddr[P]=NumAddr[P]+InCnt;
    mdix[P]=(mdix[P]+1)%Mdeg;
   }
   Preidx[P]=idx[P];
   tag.integer=tag_lst[P][idx[P]];
   setno.integer=setno_lst[P][idx[P]];
   wdno.integer=wdno_lst[P][idx[P]];
   rw.symbol=rw_lst[P][idx[P]];
   p_tag=P_makevalue(INTEGER, tag);
   p_setno=P_makevalue(INTEGER, setno);
   p_wdno=P_makevalue(INTEGER, wdno);
   p_rw=P_makevalue(SYMBOL, rw);
   p_addr=P_maketuple(p_tag, p_setno, p_wdno, 0);
   q=P_maketuple(p_addr, p_rw, 0);
   return q ;
 }

P_Valueptr wait_on_line_4 ()
 {
   int  P=3;          P_Atom  a_wait;        P_Valueptr  p_wait;
```

```
    a_wait.integer= ++(WaitBuf[P]) ;
    p_wait=P_makevalue(INTEGER, a_wait) ;
    return p_wait ;
  }

P_Valueptr wait_signal_4 ()
  {
    int  P=3;          P_Atom  a_wait;        P_Valueptr  p_wait;

    a_wait.integer= ++(WaitSig[P]) ;
    p_wait=P_makevalue(INTEGER, a_wait) ;
    return p_wait ;
  }


/*  The simulation provided procedures are cut here since they are lengthy and with merely minor
    difference from one another. However, they are available from the writer.
*/
```

REFERENCE C: A GRAMMAR FOR THE PAISLEY LANGUAGE

The grammar for PAISLey, first published in [Zav82], then modified in [Zav87b], is LALR. We reprint, by courtesy of P. Zave, this grammar here in BNF for convenience and completion. In the following text, bold words mean terminals or reserves, and italic words classify categories and some semantics which are unable for LALR to express.

1. BNF Productions with Constraints

Specifications and Statements

spec ::= statement-list ;

statement-list ::= statement-list ; statement | statement |
statement-list ; replicator < ; statement-list > |replicator < ; statement-list >

statement ::= system-decl | set-defn | map-decl | map-defn | timing-constraint

Replicators

replicator ::= replication-word # numeral-word .. numeral-word |
numeral-word .. numeral-word

In a replicator the second unsigned integer must be at least as great as the first. Do not put the # of a replicator in the first column of a line - that will cause an unwanted interaction with the C preprocessor. Also, in an expression such as #1.. Param (where the C preprocessor will replace Param by an integer), the space before Param is necessary due to some peculiarity of the C preprocessor.

replication-word ::= lower-word | upper-word | capitalized-word

If replications are nested then the replication words must be distinct, even without regard to case.

System Declarations and Processes

system-decl ::= (process-list)

process-list ::= process-list , process | process | process-list , replicator < , process-list > |
replicator < , process-list >

process ::= map-name [map-exp]

Set Definitions and Set Expressions

set-defn ::= set-name = set-exp

Set definitions must not be recursive.

set-exp ::= set-exp | set-term | set-item | set-exp |
replicator < | set-exp > | replicator < | set-exp >

set-term ::= set-term * set-item | set-item |
set-term * replicator < * set-term > | replicator < * set-term >

set-item ::= set-name | (set-exp) | { value-enum }

Set Names

set-name ::= upper-word - set-name-words | upper-word

A set name may not be longer than 4096 characters. It may not contain transparent text.

set-name-words ::= set-name-words - set-word | set-word

set-word ::= upper-word | numeral-word

Value Enumerations and Value Expressions

value-enum ::= value-enum , value-exp | value-exp | replicator < , value-enum > |
value-enum , replicator < , value-enum > |
value-exp ::= value-name | (value-list)
value-list ::= value-list , value-exp | value-exp |
value-list , replicator < , value-exp > | replicator < , value- exp >

Value Names

value-name ::= ' ascii-string ' | symb-value-name | integer-value | real-value

Actually the two single-quote characters (' ') may be used interchangeably. A string value may not be longer than 4096 characters.

symb-value-name ::= capitalized-word - value-name-words | capitalized-word

A symbolic-value name may not be longer than 4096 characters and must contain no transparent text.

value-name-words ::= value-name-words - value-word | value-word
value-word ::= capitalized-word | numeral-word
integer-value ::= numeral-word | sign numeral-word

An integer value contain no transparent text. Internally integer values are represented by "long" integers on the host machine, and any specified integer larger than the capacity of that representation is not acceptable.

real-value ::= real-string | sign real-string | real-string **E** integer-value |
sign real-string **E** integer-value

A real value must contain no transparent text. Internally real values are represented by single precision floating-point numbers on the host machine, and any specified real value larger than the capacity of that representation is not acceptable.

sign ::= + | -
real-string ::= numeral-word . numeral-word | numeral-word . | . numeral-word

Mapping Declarations

map-decl ::= map-name : set-exp --> set-exp | map-name : --> set-exp

The arrow may also be written -> or --->.

Mapping Definitions

map-defn ::= map-name = map-exp | map-name formal-params = map-exp

formal-params ::= [param-exp]

Formal-Parameter Expressions

param-exp ::= param-name | (param-list)

param-list ::= param-list , param-exp | param-exp |
param-list , replicator < , param-exp > | replicator < , param-exp >

Formal-Parameter Names

param-name ::= lower-word - param-name-words | lower-word

A formal-parameter name may not be longer than 4096 characters. It may not contain transparent text.

param-name-words ::= param-name-words - param-word | param-word

param-word ::= lower-word | numeral-word

Mapping Expressions

map-exp ::= map-name | value-exp | map-appl | (map-list) | / pred-pair-list , **True** : map-exp

map-appl ::= map-name [map-exp] | replicator < map-name > [map-exp]

map-list ::= map-list , map-exp | map-exp |
map-list , replicator < , map-exp > | replicator < , map-exp >

pred-pair-list ::= pred-pair-list , pred-pair | pred-pair |
pred-pair-list , replicator < , pred-pair-list > | replicator < , pred-pair-list >

pred-pair ::= map-exp : map-exp

Mapping Names

map-name ::= lower-word - map-name-words | lower-word

A mapping name may not be longer than 4096 characters. It may not contain transparent text.

map-name-words ::= map-name-words - map-word | map-word

map-word ::= lower-word | numeral-word

Timing Constraints

timing-constraint ::= map-name : ! attribute-list

attribute-list ::= attribute-list , attribute | attribute

attribute ::= real-attribute | distribution-attribute | proportion-attribute

real-attribute ::= real-attribute-name real-value time-unit

real-attribute-name ::= lb | ub | mean | stdev | mean-1 | mean-2

All real values in real attributes must be positive except those quantifying lb attributes, which must be nonnegative. If a mapping has both lb and ub attributes, the upper bound must be greater than or equal to the lower bound.

distribution-attribute ::= uniform | normal | exponential | hyperexponential

A mapping with a uniform distribution must also have lb and ub attributes, but no others. A mapping with a normal distribution must also have mean and stdev attributes. It may also have lb and ub attributes, but no others. A mapping with an exponential distribution must also have a mean attribute. It may also have lb and ub attributes, but no others. A mapping with a Hyperexponential distribution must also have mean-1, mean-2, and proportion-1 attributes. It may also have lb and ub attributes, but no others.

proportion-attribute ::= proportion-1 real-value

The real value in a proportion attribute must lie between zero and one.

time-unit ::= **hr | m | s | ms | us | ns**

Full spellings and other abbreviations of the time units are also acceptable.

2. Lexical Primitives

ascii-string ::= any string of ASCII characters with length greater than or equal to zero

upper-word ::= any string of upper-case alphabetical characters with length greater than zero

lower-word ::= any string of lower-case alphabetical characters with length greater than zero

capitalized-word ::= any string of alphabetical characters, with length greater than zero, whose first letter is upper-case and all of whose remaining letters are lower-case

numeral-word ::= any string of numerals with length greater than zero

comment ::= any ascii-string enclosed in double-quote characters

transparent-text ::= a comment, blank, carriage-return, line-feed, or tab character.

Transparent-text is transparent anywhere except within a string value, and allowed without altering any semantics everywhere except where specifically prohibited.

3. Summary of Intrinsic

List of Intrinsic Sets in Alphabetical Order

ANY = the set of all representable values

BOOLEAN = { True, False }

FILLER = { Null }

INTEGER = the set of all representable integer value

REAL = the set of all representable real values

STRING = the set of all representable string value

List of Intrinsic Mappings in Alphabetical Order

abs: INTEGER --> INTEGER

and: BOOLEAN * BOOLEAN --> BOOLEAN

cat: STRING * STRING --> STRING

diff: INTEGER * INTEGER --> INTEGER

equal: ANY * ANY --> BOOLEAN

ex-or: BOOLEAN * BOOLEAN --> BOOLEAN

g-t: INTEGER * INTEGER --> BOOLEAN

g-t-e: INTEGER * INTEGER --> BOOLEAN

i-to-r: INTEGER --> REAL

i-to-s: INTEGER --> STRING

is-null: ANY --> BOOLEAN

l-index: STRING * STRING --> INTEGER

l-t: INTEGER * INTEGER --> BOOLEAN

l-t-e: INTEGER * INTEGER --> BOOLEAN

len: STRING --> INTEGER

mod: INTEGER * INTEGER --> INTEGER

not: BOOLEAN * BOOLEAN --> BOOLEAN

or: BOOLEAN * BOOLEAN --> BOOLEAN

prod: INTEGER * INTEGER --> INTEGER

proj: INTEGER * ANY --> ANY

quot: INTEGER * INTEGER --> INTEGER

r-abs: REAL --> REAL

r-diff: REAL * REAL --> REAL

r-g-t: REAL * REAL --> BOOLEAN

r-g-t-e: REAL * REAL --> BOOLEAN

r-index: STRING * STRING --> INTEGER

r-l-t: REAL * REAL --> BOOLEAN

r-l-t-e: REAL * REAL --> BOOLEAN

r-prod: REAL * REAL --> REAL

r-quot: REAL * REAL --> REAL

r-sum: REAL * REAL --> REAL

r-to-i: REAL --> INTEGER

r-to-s: REAL --> STRING

replace: ANY * INTEGER * ANY --> ANY

s-g-t: STRING * STRING --> BOOLEAN

s-g-t-e: STRING * STRING --> BOOLEAN

s-l-t: STRING * STRING --> BOOLEAN

s-l-t-e: STRING * STRING --> BOOLEAN

s-to-i: STRING --> INTEGER

s-to-r: STRING --> REAL

sq-rt: REAL --> REAL

substr: STRING * INTEGER * INTEGER --> STRING

sum: INTEGER * INTEGER --> INTEGER

x-map-name-words: ANY --> ANY

xm-map-name-words: ANY --> ANY

xr-map-name-words: ANY --> ANY

APPENDIX D: THE PAISLEY INTERPRETER COMMANDS

For the referencing convenience, syntax of the PAISLey interpreter command language is also copied here. A minor change is made from the original notations to follow the convention of BNF. Words in Times Roman are terminals and words in italic are nonterminals. The nonterminals "*integer-value*," "*real-value*," and "*time-unit*" are defined in the grammar of the previous appendix. The nonterminal "*file-name*" refers to any valid UNIX file name. An "*integer-list*" refers to a list of integer values delimited by spaces. Similarly, a "*mapping-name-list*" refers to a list of the PAISLey's mapping names, a "*channel-name-list*" refers to a list of channel names, and all members of these lists are separated by spaces. Any one of the keywords can be used if they are enclosed in a list, and separated by "|", meaning "or". The keywords enclosed in angle brackets means using one of these keywords is mandatory, whereas the keywords enclosed in braces are optional. The semantics of the interpreter commands can be further referenced in the PAISLey's user manuals [Zav87a, 87b].

Global control commands

help

start

restart

continue

quit

Historical information saving and using commands

save *file-name*

settrace on *file-name*

settrace off

script *file-name*

Trace formatting commands

setlimit *integer-value*[*]

setwidth *integer-value*

setcolumns *integer-list*

setreport < initiation | terminate | both > < on | off > *mapping-name-list*

settrap < initiation | terminate | both > < on | off > *mapping-name-list*

Incremental testing control commands

setmode < default | interactive | random > *mapping-name-list*

setprocess < on | off > *integer-list*

setchannel < on | off > *channel-name-list*

Simulation control commands

setbtu [*real-value* | *integer-value*] *time-unit*[**]

setoverhead [*real-value* | *integer-value*] *time-unit*

setscheduling < topdown | bottomup > *integer-list*

setscale < *real-value* | *integer-value* > *integer-list*

---

[*] Zero refers no man-made limit, but subject to the physical limitation of resources.

[**] The legal basic time units are hr (hour), m (minute), s (second), ms (millisecond), us (microsecond) and ns (nanosecond). The default is one millisecond.

BIBLIOGRAPHY

[AgaS88] Agarwal, R. R., Simoni, R., Hennessy, J. and Horowitz, M., An evaluation of directory schemes for cache coherence. 15th Ann. Int'l. Symp. on Computer Architecture, (1988), 280-289.

[Agn91] Agnew, D., VHDL extensions needed for synthesis and design. Proc. IFIP on CHDL and Their Applications. Edited by Borrione, D. and Waxman, R., 1991, 335-349.

[AhtA86] Ahtiainen, A, Alfonzetti, S., Chari, V., *et al.*, An approach for evaluating formal description techniques. Protocol Specification, Testing, and Verification. Edited by Diaz, M. V., North-Holland, 1986.

[AmbH85] Amblard, A., Caspi, P., Halbwachs, N., Describing and reasoning about circuits behavior by means of time functions. Proc. IFIP 10.2 on CHDLs and Their Applications, Tokyo, Japan, Aug. 1985, 39-47.

[ArcB86] Archibald, J. and Baer, J. L., Cache coherence protocols: evaluation using a multiprocessor simulation model. ACM Trans. on Computer Systems, Vol. 4, No. 4 (Nov. 1986), 273-298.

[ArmR87] Armstrong, J.R., Roumeliotis, M., HDL modeling for process oriented simulation. Proc. IFIP on CHDL and Their Applications, Amsterdam, The Netherlands, April 1987, 1-8.

[Ayl86] Aylor, J. H., VHDL-feature description and analysis. IEEE Design & Test, April 1986, 17-27.

[Bac78] Backus, J., Can programming be liberated from the von neumann style? - A functional style and its algebra of programs. Commun. ACM, Vol. 21, No. 8 (Aug., 1978), 613-641.

[Bae80] Baer, J., Computer Systems Architecture. Computer Science Press, Inc., 1980.

[Bak88] Bakowski, P., Design analysis at RT level using correlated representation methods. Tool Integration and Design Environments. Edited by Rammig, F. J., 1988.

[Barb75] Barbacci, M. R., A comparison of register transfer languages for describing computers and digital systems. IEEE Trans. Computers, Vol. C-24, No. 22 (Feb. 1975), 137-150.

[Barb81] Barbacci, M. R., Instruction set processor specifications (ISPS): the notation and its applications. IEEE Trans. Computers, Vol. C-30, No. 1 (Jan. 1981), 24-40.

[Barb82] Barbacci, M. R. and Siewiorek, D.P., The Design and Analysis of Instruction Set Processors. McGraw-Hill, Inc., New York, 1982.

[BarbG85] Barbacci, M. R., Grout, S., *et al.*, Ada as a hardware description language: an initial report. Proc. IFIP 10.2 on CHDLs and Their Applications, Tokyo, Japan, Aug. 1985.

[BarbK87] Barbacci, M. and Koomen, K., Ed., Computer Hardware Description Languages and Their Applications. North-Holland, 1987.

[BarbU85] Barbacci, M. R. and Uehara, T., Guest Editors, Special issue on CHDLs: the bridge between software and hardware. IEEE Computer, Vol. 18, No. 2 (Feb. 1985), 6-8.

[Bart88] Barton, D. L., A first course in VHDL. Intermetrics Inc., Bethesda, MD, Design Automation Guide 1988.

[BecO89] Beck, B. and Olien D., A parallel-programming process model. IEEE Software (May 1989) 63-72.

[Bel71] Bell, C. G., Computer Structures: Readings and Examples. McGraw-Hill, Inc., 1971.

[BelB77] Bell, T. E., Bixler, D. C. and Dyer, M. E., An extendable approach to computer-aided software requirements engineering. IEEE Trans. Softw. Eng., Vol. SE-3, No. 1 (Jan. 1977), 49-60.

[BergZ86] Bergland, G. D. and Zave, P., Special issue on software design methods. IEEE Trans. Softw. Eng. Vol. SE-12, No. 2 (Feb. 1986), 185-191.

[BerlZ87] Berliner, E. F. and Zave, P., An experiment in technology transfer: PAISLey specification of requirements for an undersea lightwave cable system. Proc. Ninth Int. Conf. Software Engineering, Montery, California, March 1987, 42-50.

[Boc82] Bochmann, G. V., Hardware specification with temporal logic: an example. IEEE Trans. Comp., Vol. C-31, No. 3 (March 1982), 223-231.

[BorF85] Borrione, D. and Faou, C. L., Overview of the CASCADE multi-level hardware description language and its mixed-mode simulation mechanisms. Proc. IFIP 10.2 on CHDLs and Their Applications, Tokyo, Japan, Aug. 1985, 239-260.

[BorW91] Borrione, D. and Waxman, R., Computer Hardware Description Languages and Their Applications. Proc of the 10th IFIP. Symp., North-Holland, 1991.

[Bou91] Boute, R. T., Declarative language - still a long way to go. Proc. IFIP on CHDL and Their Applications. Edited by Borrione, D. and Waxman, R., 1991, 185-212.

[BraB68] Branes, G. H., Brown, R. M., *et al.*, The ILLIAC IV computer. IEEE Trans. Computers, Vol. C-18, No. 8 (Aug. 1968), 746-757.

[BroC85] Browne, M., Clarke, E. M., Dill, D., Mishra, B., Automatic verification of sequential circuits using temporal logic. Proc. IFIP 10.2 on CHDLs and Their Applications,

Tokyo, Japan, Aug. 1985.

[Bru86] Brunes, G. R., Technology Assessment: PAISLey. Tech. Rep. STP-296-86, Microelectronics and Computer Technology Corp., Austin, Texas, Sept. 1986.

[BruK87] Bruck, R., Klomps, R. and Schuetz, J., Analysis of behavioral DACAPO descriptions. Proc. IFIP on CHDL and Their Applications, Amsterdam, The Netherlands, April 1987.

[BruM86] Bruno, G. and Marchetto, G., Process-translatable petri nets for the rapid prototyping of process control systems. IEEE Trans. Softw. Eng., Vol. SE-12, No. 2 (Feb, 1986).

[Bul87] Bulterman, D. C. A., An animated modelling environment for parallel architectures. Proc. IFIP on CHDL and Their Applications, Amsterdam, The Netherlands, April 1987, 115-127.

[BurM89] Burkhart, H. and Millen, R., Performance-measurement tools in a multiprocessor environment. IEEE Trans. Computers, Vol. C-38, No. 5 (May 1989), 725-737.

[Cam89] Camposano, R., Behavior-preserving transformations for high-level synthesis. Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects, July 5-7, 1989, Cornell Univ., Ithaca, New York.

[Chu65] Chu, Y., An Argol-like computer design language. Comm. Ass. Comput. Math., Vol. 8 (Oct. 1965), 605-615.

[Chu72a] Chu, Y., Computer Organization and Microprogramming. Prentice Hall, Englewood Cliffs, New Jersey, 1972.

[Chu72b] Chu, Y., Introduction of the computer design language. Proc. IEEE Comput. Conf., COMPCON 72, San Francisco, Calif, Sept. 1972, 215-218.

[Chu74] Chu, Y., Guest Editor, Special issue on CHDLs. IEEE Computer, Vol. 7, NO. 12 (Dec. 1974), 18-22.

[CoeM91] Coen-Porisini, A., Morzenti, A., *et al.*, Specification and verification of hardware systems using the temporal logic language TRIO. Proc. IFIP on CHDL and Their Applications. Edited by Borrione, D. and Waxman, R., 1991, 43-61.

[DaeH85] Daeche, N. and Hanna, F. K., Specification and verification using higher-order logic. Proc. IFIP 10.2 on CHDLs and Their Applications, Tokyo, Japan, Aug. 1985.

[Dam85] Damm, W., Design and specification of microprogrammed computer architecture. Proc. 18th Annual Workshop on Microprogramming, IEEE/CS Press, Los Alamitos, Calif., 1985, 3-10.

[DamD86] Damm, W., Doehmen, G., Merkel, K., *et al.*, The AADL/S* approach to firmware design verification. IEEE Software, Vol. 3, No. 4, (April 1986) 27-37.

[Dan87] Daniels, M., Design criteria and formal description techniques. Proc. IFIP on CHDL

and Their Applications, Amsterdam, The Netherlands, April 1987.

[Dar69] Darringer, J. A., The Description, Simulation and Automatic Implementation of Digital Computer Processor. Ph.D. Dissertation, Dept. Elec. Eng., Carnegie-Mellon Univ., Pittsburgh, PA. May, 1969.

[DarR89] Darringer, J. A. and Rammig, F. J., Ed., Computer Hardware Description Languages and Their Applications. Proc. of the 9th IFIP Symp, June 19-21, 1989.

[Das84a] Dasgupta, S., The Design and Description of Computer Architectures. John Wiley Sons, Inc., 1984.

[Das84b] Dasgupta, S., A Model of clocked micro-architectures for firmware engineering and design automation applications. Proc. 17th Annual Microprogramming Workshop, IEEE/CS Press, 1984. 298-308.

[Das89a] Dasgupta, S., Computer Architecture - A Modern Synthesis, Vol.1: Foundation. John Wiley & Sons, Inc., 1989.

[Das89b] Dasgupta, S., Computer Architecture - A Modern Synthesis, Vol.2: Advanced Topics. John Wiley & Sons, Inc., 1989.

[DasA87] Dasgupta, S. and Aguero, U., On the plausibility of architectural design. Proc. IFIP on CHDL and Their Applications, Amsterdam, The Netherlands, April 1987, 177-193.

[DasW86] Dasgupta, S. and Wilsey P. A., Axiomatic specifications in firmware development systems. IEEE Software, Vol. SE-3, No. 7 (July 1986), 49-58.

[DavM87] Davie B. S. and Milne, G. J., The role of behavior in VLSI design languages. HDL Descriptions to Guaranteed Correct Circuit Designs, edited by Dominique Borrione, North-Holland, 1987.

[DotA90] Dotan, Y. and Arazi, B., Concurrent logic programming as a hardware description tool. IEEE Trans. Comp., Vol. C-39, No. 1 (Jan. 1990), 72-88.

[Dul70] Duley, J. R., DDL - A Digital Design Language. Ph.D. Dissertation, Dept. Elec. Eng., Univ. Wisconsin, Madison, June 1970.

[DulD68] Duley, J. R. and Dietmeyer, D. L., A digital system design language (DDL). IEEE Trans. Comput., Vol. C-17, No. 9 (Sept. 1968), 850-861.

[DutG89] Dutt, N. D. and Gajski, D. D., EXEL: A language for interactive behavioral synthesis. Proc. 9th IFIP Symp. on CHDLs and Their Applications, June 19-21, 1989, Washington, D. C., 3-18.

[Eve85] Eveking, H., The application of CHDL's to the abstract specification of hardware. Proc. IFIP 10.2 on CHDLs and Their Applications, Tokyo, Japan, Aug. 1985, 167-178.

[FasH92] Fasel, J. H. and Hudak, P., *et al.*, ACM SIGPLAN Notices Special Issue on the

Functional Programming Language Haskell. ACM Press, Vol. 22, No. 5 (May 1992).

[FilF84] Filman, R. E. and Friedman, D. P., Coordinated Computing: Tools and Techniques for Distributed Software. McGraw-Hill, New York, 1984, 79-81.

[FisR91] Fisher, J. A. and Rau, B. R., Instruction-level parallel processing. Science, Vol. 913 (Sept. 1991), 1233-1241.

[FitZ77] Fitawater, D.R. and Zave, P., The use of formal asynchronous process specifications in a system development process. Proc. of the 6th Texas Conference on Computer Systems, Austin, Texas, Nov. 1977, 2B-21 - 2B-30.

[Fra84] Frank, S. J., Tightly coupled multiprocessor systems speed memory access times. Electronics, Vol. 57, No. 1 (Jan. 1984), 164-169.

[FujF89] Fujita, M and Fujisawa, H., Specification, verification and synthesis of control circuits with propositional temporal logic. Proc. 9th IFIP Symp. on CHDLs and Their Applications, June 19-21, 1989, Washington, D. C., 265-279.

[FujT85] Fujita, M., Tanaka, H., Moto-oka, T., Logic design assistance with temporal logic. Proc. IFIP 10.2 on CHDLs and Their Applications, Tokyo, Japan, Aug. 1985.

[GerK87] Gertner, I. and Kurshan, R. P., Logic analysis of digital circuits. Proc. IFIP on CHDL and Their Applications, Amsterdam, The Netherlands, April 1987, 47-67.

[GheM90] Ghezzi, C., Mandrioli, D. and Morzenti, A., TRIO, a logic language for executable specifications of real-time systems. Journal of Systems and Software, Vol. 12, No. 2 (May 1990), 107-123.

[Goe92] Goering, R., VHDL's missing link: the gate level. Electronic Engineering Times, Jan. 6, 1992, 27-28.

[Goo83] Goodman, S. J., Using cache memory to reduce processor-memory traffic. 10th Ann. Int'l. Symp. on Computer Architecture, 1983, 124-131.

[Goo87] Goodman, J. R., Cache memory optimization to reduce processor/memory traffic. J. VLSI Comput. Syst., Vol. 22, No. 2 (1987), 61-86.

[GorM87] Gordon, M., Melham, T. and Camilleri, A., Hardware verification using higher-order logic. HDL Descriptions to Guaranteed Correct Circuit Designs. Edited by Dominique Borrione, North-Holland, 1987, 43-67.

[Got83] Gottlieb, A., The NYU ultracomputer-designing an MIMD shared-memory parallel computer. IEEE Trans. Computers, Vol. C-32, No.2 (Feb. 1983), 175-189.

[Hab89] Habib, S., Microprogrammed architectures specified using PAISLey. ACM SIGMICRO Newsletter, Vol. 20, No. 2 (June 1989) 15-18.

[Hab91] Habib, S., A brief walk in microprogramming technologies, CISC, RISC and VLIW. Seminar in Computer Architecture: EPFL, Lausanne, Switzerland, Oct. 1991.

[HabJ90] Habib, S. and Jin, T., Computer Hardware Systems Description Using Operational Specification. Technical report of the Research Foundation of CUNY, No. 6-61330. January, 1990.

[HabJ92a] Habib, S. and Jin, T., Comparison of the operational applicative high order logic as a CHDL with VHDL. VHDL Methods Workshop, Charlottesville, Virginia, August 10-12, 1992.

[HabJ92b] Habib, S. and Jin, T., Functional and operational specifications of the multiprocessor system using PAISLey. Submitted for publication.

[Hal87] Hale, R., Temporal logic programming. Temporal Logic and Their Applications. Edited by Galton, A., Academica Press, 1987.

[HanD89] Hanna, F. K., Daeche N. and Longley, M., VERITAS+: a specification language based on type theory. Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects, July 5-7, 1989, Cornell Univ., Ithaca, New York.

[Hare87] Harel, D., STATECHARTS: A visual formalism for complex systems. Science of Computer Programming, Aug. 1987, 231-274.

[HareL88] Harel, D., Lachover, H., *et al.*, STATEMATE: A working environment for the development of complex reactive systems. Proc. 10th Int. Conf. Softw. Eng., April 1988, Singapore, 396-406.

[Hart87a] Hartenstein, R. W., Ed., Hardware Description Languages. North-Holland, 1987.

[Hart87b] Hartenstein, R. W., The classification of hardware description languages. Hardware Description Languages. Edited by the same person, North-Holland, 1987, 15-47.

[Hay88] Hayes, J. P., Computer Architecture and Organization, 2nd Ed. McGraw-Hill Book Company, 1988.

[HenP90a] Hennessy, J. L. and Patterson, D. A., Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, Inc. 1990.

[HenP90b] Hennessy, J. L. and Patterson, D. A., Software Distribution for Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, Inc., 1990.

[Her85] Herbert, J., The application of formal specification and verification to a hardware design. Proc. IFIP 10.2 on CHDLs and Their Applications, Tokyo, Japan, Aug. 1985.

[HilP78] Hill, F. J. and Peterson G. R., Digital Systems: Hardware Organization and Design. Wiley, New York, 1978.

[HsiP85] Hsieh, J. T., Pleszkun, A. R., Vernon, M. K., Performance evaluation of a pipelined VLSI architecture using the graph model of behavior (GMB). Proc. IFIP 10.2 on CHDLs and Their Applications, Tokyo, Japan, Aug. 1985, 192-205.

[Hun87] Hunt, W. A. Jr, The mechanical verification of a microprocessor design. HDL

Descriptions to Guaranteed Correct Circuit Designs. Edited by Dominique Borrione, North-Holland, 1987.

[IEE87] IEEE Standard 1076-1987, VHDL Language Reference Manual.

[Ive62a] Iverson, K. E., A common language for hardware, software, and applications. Proc. Fall Joint Comput. Conf., 1962, 121-129.

[Ive62b] Iverson, K. E., A Programming Language. Wiley, New York, 1962.

[Jal89] Jalote, P., Testing the completeness of specifications. IEEE Trans. Softw. Eng., Vol. SE-15, No. 5 (May 1989), 626-531.

[JinH92] Jin, T. and Habib, S., Computer system specification using applicative high order logic. Submitted for publication.

[Kin85] Kinoshita, K., Testable design and design language. Proc. IFIP 10.2 on CHDLs and Their Applications, Tokyo, Japan, Aug. 1985, 21-29.

[Lar87] Larsson, T., Specification and verification of VLSI systems actional behavior. Proc. IFIP on CHDL and Their Applications, Amsterdam, The Netherlands, April 1987, 267-279.

[LeeS88] Lee, C., Skedzielewski, S. and Feo, J., On the implementation of applicative languages on shared-memory, MIMD multiprocessor. Proc. ACM/SIGPLAN PPEALS 1988, New Haven, Connecticut. July 1988, 188-197.

[Lie85] Lieberherr, K. J., Toward a standard hardware description language. IEEE Design and Test, Feb. 1985, 55-62.

[LieK83] Lieberherr, K. J. and Knudsen, S. E., Zeus: A hardware description language for VLSI. ACM/IEEE Proc. 20th Design Automation Conf., June 27-29, 1983, 17-23.

[LippM88] Lipper, E. H., Melamed, B., Morris, R. J. T. and Zave, P., A multi-level secure message switch with minimal TCB: architectural outline and security analysis. Proc. 14th Aerospace Computer Security Applications Conf., 1988, 242-249.

[LipsM86] Lipsett, R., Marschner, E, and Shahdad, M., VHDL - the language. IEEE Design & Test, April 1986, 28-41.

[MaiO82] Maissel, L. I. and Ostapko, D. L., Interactive design language: a unified approach to hardware simulation, synthesis, and documentation. 19th Design Automation Conf., Las Vegas, Nev., June 1982, 193-201

[MaiO84] Maissel, L. I. and Ofek, H., Hardware design and description languages in IBM. IBM J. Research and Development, Vol. 26, No.5 (Sept. 1984), 557-563.

[MaiP83] Maissel, L. I. and Phoenix, R. L., IDL (Interactive Design Language) features and philosophy. IEEE Int'l Conf. Computer Design, Port Chester, NY, Sept. 1983, 667-669.

[MarF85] Maruyama F. and Fujita M., Hardware verification. IEEE Computer, Vol. 18, No. 2 (Feb. 1985), 22-32.

[McCZ89] McCoy, E. E. and Zave, P., Multistage, multifaced analysis of telecommunications network architectures. Proc. IEEE Infocom '89. IEEE Computer Society Press, Washington, D. C., 1989, 91-95.

[Mes82] Meszleny, C. K., Computer Design Language Simulation and Boolean Translation. Comput. Sci. Center, Univ. Maryland, College Park, Tech. Rep., June 1982, 68-72.

[Mil85] Milne, G. J., Simulation and verification: related techniques for hardware analysis. Proc. 7th IFIP Symp. on CHDLs and Their Applications, Tokyo, Japan, Aug. 1985, 404-417.

[MitF90] Mitchell, C. L. and Flynn, M. J., The effects of processor architecture on instruction memory traffic. ACM Trans. Computer Systems, Vol. 8, No. 3 (Aug. 1990), 230-250.

[Mos84] Moszkowski, B., Executing temporal logic program. LNCS 164, Springer Verlag, 1984.

[Mos85] Moszkowski, B., A temporal logic for multilevel reasoning about hardware. IEEE Computer, Vol. 18, No.2 (Feb. 1985), 10-19.

[MulR89] Muller, W. and Rammig, F., ODICE: Object-oriented hardware description In CAD environment. Proc. 9th IFIP Symp. on CHDLs and Their Applications, June 19-21, 1989, Washington, D. C., 19-34.

[Mye82] Myers, G. J., Advances in Computer Architecture. Wiley, New York, 1982.

[NarA88] Narayana, K. T. and Aaby, A. A., Specification of real-time systems in real-time temporal interval logic. Proc. Real-Time Systems Symp., Huntsville, Alabama, Dec 6-8, 1988, 86-95.

[NasS86] Nash, I. D. and Saunders, L. F., VHDL critique. IEEE Design & Test, April 1986, 54-65.

[Nol92] Nolan, K., Requirements for analog expressions to VHDL. VHDL Methods Workshop, Charlottesville, Virginia, August 10-12, 1992.

[Pai87] Paillet, J. L., A functional model for descriptions and specifications of digital devices. HDL Descriptions to Guaranteed Correct Circuit Designs. Edited by Dominique Borrione, North-Holland, 1987.

[ParP84] Paramarcos, M. and Patel, J., A low overhead coherence solution for multiprocessors with private cache memories. 11th Ann. Int'l. Symp. on Computer Architecture, 1984, 348-354.

[Pat76] Patterson, D. A., STRUM: Structured programming system for correct firmware. IEEE Trans. Comput., Vol. C-25, No. 10 (Oct. 1976), 974-985.

[Pat81] Patterson, D. A., An experiment in high level language microprogramming and verification. Comm. ACM, Vol. 24, No. 10 (Oct. 1981), 699-709.

[PatS81] Patterson, D. A. and Sequin, C. H., RISC I: A Reduced Instruction Set VLSI Computer. Proc. 8th Annual Int. Symposium on Comp. Arch., IEEE Computer Society Press, 1981, 443-458.

[PatS82] Patterson, D. A. and Sequin, C. H., A VLSI RISC. IEEE Computer Vol. 15, No. 9 (Sept. 1982), 8-21.

[PawW87] Pawlak, A. and Wrona, W., Modern object_oriented programming language as an HDL. Proc. IFIP on CHDL and Their Applications, Amsterdam, The Netherlands, April 1987, 343-362.

[Per78] Per Brinch Hansen, Distributed processes: a concurrent programming concept. Comm. ACM, Vol. 21, No. 11 (Nov. 1978), 934-941.

[Pia84] Piatz, S. J., HSL: a comprehensive hardware design language. Proc. 17th Annual Hawaii Int. Conf. on System Sc., 1984, 137-142.

[Pil80a] Piloty, R., *et al.*, CONLAN - a formal construction method for hardware description languages: basic principles, languages derivation, language application. NCC 1980 Proc., Vol. 49, 209-236.

[Pil80b] Piloty, R., *et al.*, An overview of CONLAN - a formal construction method for hardware description languages. Proc. IFIP Congress, Oct. 1980, 199-204.

[Pre91] Prete, C. A., RST cache memory design for a tightly coupled multiprocessor system. IEEE Macro, Vol. 11, No. 2 (April 1991), 16-19, 40-52.

[PrzH89] Przybylski, S., Horowitz, M. and Hennessy, J., Characteristics of performance-optimal multi-level cache hierarchies. 16th Ann. Int'l. Symp. on Computer Architecture, 1989, 114-121.

[RamS80] Ramamoorthy, C.V. and So, G.S., Performance evaluation of asynchronous concurrent systems using petri nets. IEEE Trans. Softw. Eng. Vol. SE-6, No. 9 (Sept. 1980), 440-449.

[Ros77] Ross, D. T., Structured analysis (SA): a language for communicating ideas. IEEE Trans. Softw. Eng., Vol. SE-3, No. 1 (Jan. 1977), 16-34.

[SadV88] Sadayappan, P. and Visvanthan V., Circuit simulation on shared-memory multiprocessors. IEEE Trans. Computers, Vol. C-37, No. 12 (Dec. 1988), 1634-1642.

[Sch92] Schoen, J. M., Ed., Performance and fault modeling with VHDL. Prentice Hall, Inc., 1992.

[Seg89] Segal, R. B., BDSYN Users' Manual (Ver. 1.1). Univ. of California, Berkeley, Tech. Report, March 1989.

[ShaL85] Shahdad, M., Lipsett, R., *et al.*, VHSIC hardware description language. IEEE Computer, Vol. 18, No. 2 (Feb. 1985), 94-103.

[Sin81] Sint, M., MIDL - a microinstruction description language. Proc. 14th Annual Microprogramming Workshop, 1981, 95-106.

[SmiB80] Smith, C. and Browne, J.C., Aspects of software design analysis: concurrency and blocking. Proc. of Performance '80, Toronto, Ontario, May 1980, 245-253.

[Smi82] Smith, A. J., Cache memories. Computing Surveys, Vol. 14, No. 3 (Sept. 1982), 473-530.

[Smo82] Smoliar, S. W., Applicative and functional programming. Handbook of Software Engineering. Edited by Ramamoorthy, C. V. and Vick, C. R., Prentice-Hall, Englewood Cliffs, New Jersey, 1982, 565-597.

[Sok91] Sokolowski, S., Applicative High Order Programming - the Standard ML Perspective. Chapman & Hall Computing, Lonton, 1991.

[Sto84] Stone, H. S., Database applications of the fetch-and-add instruction. IEEE Trans. Computers. Vol. C-33, No. 7 (July 1984), 604-612.

[Sto87] Stone, H. S., High-Performance Computer Architecture. Addison-Wesley Publishing Company, 1987.

[Su77] Su, S. Y. H. Su, Guest Editor, Special Issue on CHDLs. IEEE Computer, Vol. 10, No. 6 (June 1977), 10-13.

[SurI86] Surette, M. and Ingle, A., Experiences using PAISLey, a system for operational specifications. Proc. Bellcore Software Engineering Symp., Bell Communications Research, Somerset, New Jersey, Sept. 1986, 15-24.

[Szy72] Szygenda, S. A., TEGAS2 - anatomy of a general purpose test generation and simulation system. Proc. Ninth Design Automation Conf., Dallas, Texas, June 1972, 117-127.

[Tay80] Taylor, B., Introduction real time constraints into requirements and high level design of operating systems. Proc. of National Telecomm. Conference, Houston, Texas, 1980, 18.5.1-18.5.5.

[Ten76] Tennent, R. D., The denotational semantics of programming languages. Comm. ACM, Vol. 19, No. 8 (Aug. 1976), 437-453.

[UehS83] Uehara, T. and Saito, T., Maruyama, F., *et al.*, DDL verifier and temporal logic. Proc. 6th Int. Symp. CHDLs and Their Applications, North-Holland, Amsterdam, 1983, 91-102.

[WanB89] Wang, W., Bare, J. and Levy, H. M., Organization and performance of a two-level virtual-real cache hierarchy. 16th Ann. Int'l. Symp. on Computer Architecture, 1989, 140-148.

[Wad81] Wadler, P., Applicative style programming, program transformation, and list operators. Proc. Conference on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire, October, 1981, 25-32.

[Weg72] Wegner, P., Operational semantics of programming languages. ACM Conference on Proving Assertions About Programs, Las Cruces, January 1972, 128-141.

[Wilk82] Wilkes, M. V., The processor instruction set. Proc. 15th Annual Workshop on Microprogramming, IEEE Computer Society Press, Oct. 1982, 3-5.

[Wils87] Wilsey, P. A., A Hardware Description Language for Multilevel Design Automation Systems. Ph.D. Dissertation, Center for Advanced Computer Studies, Univ. Southwestern Louisiana, Lafayette, LA, 1987.

[Wils89] Wilsey, P. A., Computer architecture specification with interval temporal logic. Proc. 9th IFIP Symp. on CHDLs and Their Applications, June 19-21, 1989, Washington D. C., 35-46.

[WilsD89] Wilsey, P. A. and Dasgupta, S., Functional and operational specifications of computer architectures. Proc. 9th IFIP Symp. on CHDLs and Their Applications, June 19-21, 1989, Washington D. C., 209-224.

[WilsD90] Wilsey, P. A. and Dasgupta, S., A formal model of computer architectures for digital system design environments. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, Vol. 9, No. 5 (May 1990), 473-486.

[YehZ80] Yeh, R. T. and Zave, P., Specifying software requirements. IEEE Proc., Vol. 68, No. 9 (Sep. 1980), 1077-1085.

[Zav76] Zave, P., On the formal definition of processes. International Conference on Parallel Processing, Waldenwoods, Michigan, Aug. 1976, 35-42.

[Zav79a] Zave, P., A comprehensive approach to requirements problems. COMPSAC, Chicago, Illinois, Nov., 1979, 117-122.

[Zav79b] Zave, P., Formal specifications of complete and consistent performance Requirements. 8th Texas Conference on Computing Systems, Dallas, Texas, Nov. 1979, 4B-18 - 4B-25.

[Zav80] Zave, P., 'Real-world' properties in the requirements for embedded systems. Proc. 19th Annual Washington, D.C. ACM Technical Symp., Gaithersburg, Maryland, June 1980, 21-26.

[Zav82a] Zave, P., An operational approach to requirements specification for embedded systems. IEEE Trans. Softw. Eng., Vol. SE-8, No. 3 (May 1982), 250-269.

[Zav82b] Zave, P., Testing incomplete specifications of distributed systems. ACM/SIGACY-SIGOPS Symp. Princ. Distributed Comput. Ottawa, Canada, Aug. 1982, 42-48.

[Zav83] Zave, P., Operational specification languages. ACM Annual Conference, New York,

New York, Oct. 1983, 214-222.

[Zav84a] Zave, P., The operational versus the conventional approach to software development. Commun. ACM, Vol. 27, No. 2 (Feb. 1984), 104-118.

[Zav84b] Zave, P., An overview of the PAISLey Project - 1984. AT&T Bell Lab., Murray Hill, New Jersey, Tech. Memo., 1984.

[Zav85] Zave, P., A distributed alternative to finite-state-machine specifications. ACM Trans. Prog. Lan. and Sys., Vol. 7, No.1 (Jan. 1985), 10-36.

[Zav87a] Zave, P., PAISLey User Documentation, Volume 1, Version 3.a: Reference Manual. AT&T Bell Laboratories 11254-870904-13TM, Murray Hill, New Jersey, September 1987. (Available from author)

[Zav87b] Zave, P., PAISLey User Documentation, Volume 2, Version 3.b: Tutorial. AT&T Bell Laboratories 11254-870727-09TM, Murray Hill, New Jersey, July 1987. (Available from author)

[Zav87c] Zave, P., PAISLey User Documentation, Volume 3, Version 3.b: Case Studies. AT&T Bell Laboratories 11254-870713-07TM, Murray Hill, New Jersey, July 1987. (Available from author)

[Zav89] Zave, P., A compositional approach to multiparadigm programming. IEEE Software, Vol. 6, No. 5 (Sept. 1989), 15-25.

[Zav91] Zave, P., An insider's evaluation PAISLey. IEEE Trans. on Software Eng., Vol. SE-17, No. 3 (March 1991), 212-225.

[ZavC83] Zave, P. and Cole, G.E. Jr., A quantitative evaluation of the feasibility of, and suitable hardware architectures for, an adaptive, parallel finite-element system. ACM Trans. Math. Software, Vol. 9 (Sept., 1983), 271-292.

[ZavS86] Zave, P. and Schell, W., Salient features of an executable specification language and its environment. IEEE Trans. Softw. Eng., Vol. SE-12, No. 2 (Feb. 1986), 312-325.

[ZavY81] Zave, P. and Yeh, R. T., Executable requirements for embedded systems. Proc. 5th Int. Conf. on Softw. Eng., San Diego, California, March 1981, 295-304.