Hybrid Memory Management for Parallel Execution of Prolog
on Shared Memory Multiprocessors

Copyright © 1990 by Tam Minh Nguyen

This research is sponsored by the Defense Advanced Research Projects Agency (DARPA) (monitored by the Office of Naval Research under Contract No. N00014-88-K-0579), the NCR Corporation in Dayton, Ohio, and the National Science Foundation.

Hybrid Memory Management for Parallel Execution of Prolog on Shared Memory Multiprocessors

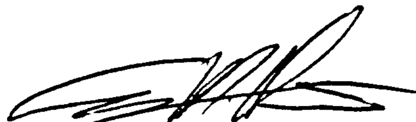
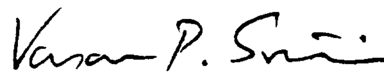
Tam Minh Nguyen

PH.D. DISSERTATION **CS DIVISION (EECS)**

ABSTRACT

Shared memory multiprocessors can provide high processing power at relatively low cost. In contrast to message passing systems, shared memory multiprocessors allow for efficient data sharing, and thus are more suitable for execution models that exploit medium grain parallelism. This dissertation investigates the problem of memory management for a globally shared space in a parallel execution environment. An AND/OR parallel execution model of Prolog is chosen for our work due to its medium grain parallelism and its intensive memory usage characteristics. With respect to space, we propose a hybrid heap-stack (called **ELPS**) which is dynamically allocated for more efficient space sharing and interference-free parallel execution. With respect to time, we present a two-tier memory architecture (called the **Aquarius-II**) with separate synchronization and high-bandwidth memory spaces.

A multiprocessor simulation system has been developed to evaluate the performance of ELPS and the Aquarius-II. ELPS incurs an average of 2% overhead (11% without hardware support), while satisfying the memory requirement to keep up with the speedup potential of the parallel execution model. Compared to the single bus multiprocessor architecture, the Aquarius-II provides higher performance by reducing contention on the synchronization bus and by providing a higher memory bandwidth with a crossbar. A simple broadcast for invalidation scheme is sufficient to keep the crossbar caches consistent while maintaining good cache performance.

Alvin M. Despain
Committee Co-Chair

Vason P. Srini
Committee Co-Chair

ACKNOWLEDGEMENT

This dissertation is the culmination of my graduate school "career." Through the years, I have received a great deal of support from many people who helped make this monumental task an achievable reality. First of all, I would like to express my deepest gratitude for my research co-advisors: Professor Alvin Despain, whose continual support and multi-faceted interest in science have fueled my enthusiasm for research; and Professor Vason Srini, who is always there for me to discuss the latest computer architectural concept as well as to complain about various aspects of graduate student life. Secondly, I would like thank my other two readers: Professor Chittoor Ramamoorthy and Professor Terence Speed. Their time and effort are greatly appreciated. Thirdly, I would like to thank: Mike Carlton, for his development of the cache simulation modules; Chien Chen (my officemate) for his cooperative effort in the development of the NuSim simulator; Bruce Holmer, for the words of encouragement in the late hours of the night; Peter Van Roy, for the insightful feedbacks to my work; and Jim Wilson and Edward Wang, for being my living encyclopedias of Unix, X-windows, Lisp, and LaTeX. I would also like to thank my other dear friends and colleagues in the Aquarius group and the C.S. Division at U.C. Berkeley, with whom I have shared many enlightening discussions. They include: Glenn Adams, Philip Bitar, Gino Cheng, Ralph Haygood, Kinson Ho, Ken Rimey, Ashok Singhal, Danielle Smith, Jerric Tam, Hervé Touati, and Benjamin Zorn.

Outside of the C.S. division, I am grateful to a number of close friends that I have made over the years. They have provided me with the friendship that I needed to continue my pursuit of a graduate degree. Among them are: Ann Greyson, Merilee Lau, Nhât Nguyễn, Arthur Sato, Roger Sit, and Lịch Trân. Although my friends in the Vietnamese Student Association at U.C. Berkeley are too many to list, they have individually and collectively made a positive impact on my outlook for the future.

I am indebted to my late grandmother for the weekly letters of encouragement that kept me afloat in my first two years at Berkeley; to my parents for their unbounded love and support, and for the constant reminder of the best measures of success: good health and happiness; and last but not least, to Quê Lam, my "best friend" and companion, for lifting me high above the hurdles in the final lap.

Contents

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 Motivation: The Memory Management Problem . . . 1
1.2 The Thesis . . . 3
1.3 Research Direction . . . 3
1.4 Contributions . . . 5
1.5 Dissertation Outline . . . 6

2 Multiprocessors and Parallel Execution 7

2.1 Multiple Processor Systems . . . 7
2.1.1 Message-Based Multicomputers . . . 7
2.1.2 Shared Memory Multiprocessors . . . 8
2.2 Parallel Execution on Multiprocessors . . . 11
2.3 Prolog and Its Applications . . . 12
2.3.1 Prolog Terminology . . . 13
2.4 Parallelism in Prolog . . . 15
2.4.1 AND-Parallelism . . . 16
2.4.2 OR-Parallelism . . . 18
2.4.3 Other Types of Parallelism . . . 20
2.5 Processes and Tasks . . . 21
2.6 Chapter Summary . . . 22

3 Memory Management: Issues and Past Solutions 23

3.1 Issues in Memory Management . . . 23
3.2 Memory Management Techniques: A Historical Perspective . . . 25
3.2.1 Virtual Memory . . . 25
3.2.2 Allocation Strategies . . . 25
3.2.3 Data Organization and Memory Access Policies . . . 27
3.3 Sequential Execution of Prolog . . . 27
3.3.1 Understanding Prolog Memory Requirements . . . 27
3.3.2 The WAM Stack Model . . . 29

3.4 Parallel Execution of Prolog . . . 31
3.4.1 Memory Requirements . . . 31
3.4.2 Scheduling Effects on Memory Behavior . . . 33
3.4.3 Message-Based Models . . . 33
3.4.4 Shared Memory Models . . . 35
3.4.5 Or-Parallel Binding Environments . . . 37
3.5 Static Partitioning of Globally Shared Space . . . 42
3.6 Solving the Problems of Static Partitioning . . . 42
3.6.1 Virtual Memory . . . 42
3.6.2 Garbage Collection . . . 43
3.6.3 Copy When Overflow . . . 44
3.6.4 Dynamic Allocation . . . 44
3.7 Chapter Summary . . . 44

4 ELPS: The Explicitly Linked Paging Stack 46
4.1 General Model . . . 46
4.1.1 Page Partitioning . . . 46
4.1.2 Link Management . . . 47
4.2 Possible Implementations . . . 48
4.2.1 Overflow and Underflow Detection . . . 48
4.2.2 Overflow and Underflow Handling . . . 50
4.2.3 Data Access . . . 50
4.2.4 Address Comparison . . . 50
4.3 Qualitative Evaluation . . . 51
4.3.1 Advantages . . . 51
4.3.2 Challenges . . . 51
4.3.3 Elimination of Address Comparison . . . 52
4.4 Chapter Summary . . . 54

5 NuSim: A Multiprocessor System Simulator 55
5.1 Introduction . . . 55
5.2 Simulator Design Goals . . . 56
5.3 Simulation System Overview . . . 57
5.3.1 Program Transformation . . . 57
5.3.2 Design Considerations . . . 59
5.4 Module Description . . . 60
5.4.1 Assembler/Loader . . . 60
5.4.2 Command Interface . . . 61
5.4.3 Graphical Interface . . . 61
5.4.4 Main Simulation Engine . . . 61
5.4.5 Memory System . . . 64
5.5 Instrumentation . . . 65
5.6 Multi-level Debugging Facility . . . 67
5.7 xNuSim: A Graphical Interface for Multiprocessor Simulators . . . 71
5.8 Compatibility and Extendability . . . 73

5.9 Implementation of ELPS on the Simulated Multiprocessor 74
5.9.1 Software Checking 74
5.9.2 Hardware Support for Checking 75
5.10 Chapter Summary 76

6 Simulator Validation **78**
6.1 Introduction 78
6.2 Validation Methodology 78
6.3 Simulator Descriptions 80
6.3.1 VPsim 80
6.3.2 Simulator Differences 81
6.4 The Validation of NuSim 82
6.4.1 Static Code Size 83
6.4.2 Cycle Count (Simulated Time) 83
6.4.3 Simulation Cost 84
6.4.4 Operation Count 87
6.4.5 Memory Accesses 88
6.5 Chapter Summary 90

7 ELPS Simulation Experiments and Results **91**
7.1 Sequential Execution Performance 91
7.1.1 Split Environment and Choice Point Stacks 91
7.1.2 Always Trail 95
7.1.3 Put Permanent Variables on Heap 97
7.2 Parallel Execution and ELPS Performance 100
7.2.1 Execution Time Overhead 100
7.2.2 Parallelism Gained 103
7.2.3 Effect of Page Size on Performance 105
7.2.4 Allocation and Deallocation Strategies 109
7.3 Discussion 109

8 Aquarius-II: A Two-Tier Memory Architecture **112**
8.1 Introduction 112
8.2 High Performance Memory Architectures 113
8.3 The Aquarius-II Architecture 114
8.3.1 Synchronization memory 114
8.3.2 High-bandwidth Memory 115
8.3.3 High-Bandwidth Memory Cache Coherency 116
8.4 Parallel Execution of Prolog 118
8.4.1 Synchronization Characteristics 119
8.4.2 Mapping of PPP onto Aquarius-II 120
8.5 Chapter Summary 121

9 Aquarius-II Simulation Results **122**
9.1 Simulation Parameters 122
9.2 Memory Access Behavior 123
9.3 Execution Time of Single Bus vs. Two Tier 125
9.4 Parallel Execution Behavior 126
9.5 Crossbar Cache Performance 129
9.5.1 Restricted Caching 129
9.5.2 Broadcast for Invalidation 130
9.6 Discussion 133

10 Conclusion **134**
10.1 Summary and Contributions 134
10.2 Future Work 135
10.3 Concluding Remarks 138

A NuSim User's Manual **139**

Bibliography **143**

List of Tables

6.1 Benchmark Code Sizes and Descriptions 83
6.2 Cycle Count and Simulation Time 85
6.3 Logical Inference Count 87
6.4 Memory References 89

7.1 Split vs. Combined Environment/Choicepoint Stack 94
7.2 Memory Statistics for Split vs. Combined Stack 94
7.3 Always Trail vs. Selective Trail (split stacks) 96
7.4 Memory Statistics for Always vs. Selective Trail (split stacks) 96
7.5 Permanent Variables on Heap vs. on Stack (split stacks) 98
7.6 Memory Statistics for Perm. Vars. on Heap vs. on Stack (split stacks) . . . 98
7.7 Benchmark Code Sizes and Descriptions 100
7.8 Overhead of ELPS Checking and Overflow Handling 101
7.9 Behavior of ELPS Checking and Overflow Handling 102
7.10 Effect of ELPS Page Size on Execution Time 105
7.11 Effect of ELPS Page Size on Overflow Frequency 108
7.12 Effect of ELPS Page Size on Internal Fragmentation 108

9.1 Access Ratios for Shared and Local Memory Areas 123
9.2 Average Access Time for Shared and Local Memory Areas 123
9.3 Execution Time and Bus Utilization of Single Bus vs. Two Tier 125
9.4 High Bandwidth Memory Access Locality 129
9.5 Write Percentage of Local Data 131
9.6 Performance of Broadcast for Invalidation Coherency Scheme 132

List of Figures

2.1 Two Categories of Shared Memory Multiprocessors 9
2.2 Components of a Prolog Program 14
2.3 Prolog Program and Corresponding AND/OR Tree 17
2.4 AND-Parallel Tree 18
2.5 OR-Parallel Tree 19
2.6 AND-OR Parallel Tree 19

3.1 Memory Hierarchy and Storage Device Relative Speeds and Costs 24
3.2 Examples of Allocation Strategies 26
3.3 Stack Usage Behavior 30
3.4 Tasks in Global Memory Space 32
3.5 Conceptual View of a Cactus Stack 35
3.6 Stack Set per Processor Memory Model 36
3.7 Stack Set per Task Memory Model 37

4.1 Fix-sized versus ELPS Variable-sized Stacks 47
4.2 Two Link Storage Schemes for ELPS 49

5.1 Program Transformation 58
5.2 NuSim Simulator Framework 60
5.3 Task Communication 63
5.4 The Multi Architecture 65
5.5 Simulation Run with NuSim Debugger 68
5.6 Simulation Run with GDB (C-language) Symbolic Debugger 69
5.7 Multi-level Debugging with NuSim Debugger and GDB 70
5.8 A Sample Setup of the xNuSim Graphical Interface 72
5.9 Hardware Support for Out-of-Page Check 76

7.1 Disadvantage of Combined Environment/Choicepoint Stack 93
7.2 Task and Processor Parallel Execution Behavior of Boyer 104
7.3 Average Effect of ELPS Page Size on Execution Time 107

8.1 The Aquarius-II Multiprocessor Architecture 114
8.2 Multiprocessor Architecture with Caches at Each Crossbar Switchpoint . . 116

8.3 Mapping of the PPP Storage Model onto the Two-Tier Memory System . . 120

9.1 Average Access Percentages of Shared and Local Memory Areas 124
9.2 Task Run Time Behavior of Single Bus vs. Two Tier (for Quicksort) 127
9.3 Task Run Time Behavior of Single Bus vs. Two Tier (for Queens6) 128

Chapter 1

Introduction

1.1 Motivation: The Memory Management Problem

As computer systems are being used to solve more complex problems, higher level languages are devised to allow the programmers to express their solutions in more natural terms. The programs are written in ways that are closer to the thought process and further away from the tedious details and constraints that exist in every computer system. This puts greater strain on the system implementors to provide efficient support in terms of library subroutines or high-level language constructs. In some areas of applications, particularly in the area of artificial intelligence, the problems require enormous symbolic processing power for a vast amount of information, in addition to the traditional arithmetic processing power. In terms of computer architecture, symbolic processing translates to simple comparison, complex pattern matching, transfer, and storage of data. Computer systems contain a hierarchy of storage designed to minimize cost while maximizing performance. Efficient management of the available memory space allows larger programs to run in a short time. While imperative languages (e.g. Pascal, C) require the programmer to explicitly allocate and deallocate memory for dynamic data structures, functional languages (e.g. Lisp) and logic programming languages (e.g. Prolog) have automatic memory allocation which frees the programmer from the tedious details of memory management.

Prolog [CM87], a programming language based on first order predicate logic [Llo87], has found its niche in the area of natural language processing, expert systems, compiler construction, geometric modeling, and design automation. Its features include pattern matching (*unification*), natural expression of non-determinism (via *backtracking*), the single-

assignment *logical* variable, and dynamic typing. These features combine to allow programmers to express algorithms in very compact code. In addition, the simple syntax and semantics of Prolog provides a useful vehicle for expressing parallelism. Two types of parallelism that exist naturally in Prolog are *AND-parallelism*, which computes subparts of a potential solution in parallel, and *OR-parallelism*, which explores alternative solutions in parallel.

On the negative side, Prolog is very memory intensive, both in terms of frequency of memory accesses and of memory space usage. The high frequency of memory access is characteristic of symbolic processing, where large amount of data needs to be transferred and compared. This behavior is in contrast to numeric applications, where the ratio of operation time to data transfer time is higher. Execution of Prolog requires a larger memory space than other languages due to two of its features: single assignment and automatic backtracking. At the source language level, the single assignment feature does not allow rewriting of the same memory location once a value has been assigned to it, and thus a new memory location has to be used. The backtracking feature requires the saving of program state and variable bindings to be restored upon backtracking. The storage space must be allocated dynamically due to the dynamic typing.

In sequential execution, memory management is largely a garbage collection problem. Numerous garbage collection techniques have been proposed for Lisp systems, and some of this technology can be transferred over to Prolog [TH88]. The Warren Abstract Machine [War83] contains stack mechanisms that can very efficiently recover unused space upon *backtracking*, which is when a program search path is terminated and an alternative path is explored. For highly deterministic programs in which little backtracking occurs, the stacks continue to grow and garbage collection is needed [TH88].

For parallel execution, memory management takes on a new perspective. In a parallel execution model for Prolog such as the PPP Execution Model [Fag87], many *tasks* are created to traverse the multiple branches of the Prolog tree structure. Each of these tasks has its own data space for storing its intermediate results, and may read data stored in other data spaces. A global address space is needed to facilitate such extensive data sharing among the tasks. For efficiency reasons, an address should fit inside a register, and thus the global address space available to the tasks is often limited by the width of the address register in the processor. With today's VLSI technology, the address registers of most commercially available processors are typically 32-bit wide, and some are 40-bit wide.

This restriction means that the address space must be distributed more efficiently among the parallel tasks, such that each task has sufficient space for private and shared data while maintaining fast access to them.

1.2 The Thesis

This dissertation examines two aspects of memory management for parallel execution of Prolog on shared memory multiprocessors: efficient space assignment for the parallel tasks and fast access to both shared and non-shared data. The thesis to be presented in this dissertation is as follows:

- *With respect to space, a dynamically-allocated, hybrid heap-stack scheme can efficiently support the space requirements of a very large number of parallel tasks within a limited space, thus allowing the potential parallelism to be fully realized by the execution model.*

- *With respect to time, a two-tier memory architecture - which has separate synchronization memory and high-bandwidth memory - can significantly reduce the synchronization bottleneck in a shared memory multiprocessor environment.*

1.3 Research Direction

Memory management for parallel execution of Prolog must take into consideration the various levels: the language data space organization, the virtual address space of the system architecture, and the system's physical memory. For complete control over the system's architectural parameters and for ease of instrumentation, we chose a simulation approach for our studies. Compared to analytical and stochastic modeling, simulation also provides more accurate performance estimates. As part of this dissertation, a complete system simulator has been written to simulate a parallel execution model of Prolog on a multiprocessor architecture. The simulator models a *shared-memory* multiprocessor system, with VLSI-PLMs as the processing units. The VLSI-PLM [STN*88] is a high performance, single VLSI chip, processor for compiled Prolog that has been fabricated and successfully tested. The simulator also models hardware extensions to the VLSI-PLM for supporting parallel execution and memory management.

Among the various parallel execution models that have been proposed, we choose

the PPP execution model by Fagin [Fag87] for study of parallel execution of Prolog. The reasons are as follows:

- *The PPP is based on the WAM*, which is an efficient and well understood engine for sequential execution of Prolog. Much research has been done at UC Berkeley on the PLM, a special-purpose architecture for Prolog, and we have learned a great deal from our past experience. The PPP model was also developed at UC Berkeley and is well-understood here.
- *The PPP supports both AND- and OR-parallelism.* Early experience with parallel execution indicates that AND-parallelism is good for some Prolog programs while OR-parallelism is more effective for others.
- *The PPP employs a shared memory architecture.* Currently, shared memory multiprocessor architectures enjoy the greatest commercial success. Systems such as the Sequent Balance [TGF88] and Encore Multimax [WWS*89] are widely used due to their low cost/performance ratios. Shared memory systems are easier to understand and to program, and free the programmer from low level memory architecture details. Since the PPP execution model exploits "medium-grain" parallelism in Prolog, a shared memory system is necessary to minimize the communication overhead.

Initial performance results of the PPP execution model reported by Fagin in [FD87] paint a dim picture of the performance of the PPP, with little speedup obtained from the set of small benchmarks. Later results reported in his dissertation [Fag87] are more encouraging. For the PLM compiler benchmark, a speedup of 7.6 was obtainable with 11 processors. In any case, we believe that these results are inconclusive and deserve further studies because:

- The PPP creates many sleeper tasks which waste memory, and sequential execution is forced when there is no more task space available.
- As pointed out by Fagin, most of the Prolog programs in his benchmark set are small (in terms of time and space requirements) and are inherently sequential [1]. Larger benchmarks are needed for a more appropriate evaluation of the execution model.

[1] Note that it is always possible to write a computer program that is strictly sequential thus no speed up is possible on any execution model. Clearly parallel execution models are effectively only with programs that have some inherent parallelism.

- The PPP model, as described by Fagin in his dissertation, is a first cut at parallel execution, and leaves out a number of details needed for practical implementation and efficient execution. This research fills in some of those gaps, particularly in memory management. The lack of memory management in this model prevents the large benchmarks from being run in Fagin's PPP simulator. The large benchmarks, which exploits medium-grain parallelism, can potentially have the greatest performance gain in this model.

- The PPP simulator written by Fagin uses an ideal, single-cycle memory. The simulator used in this research contains realistic memory parameters for a more detailed evaluation of the execution model.

1.4 Contributions

The issues of efficient parallel execution in the PPP are similar to those of other parallel models, issues such as task creation, termination, and communication. The issue of memory management for parallel execution was not addressed in Fagin's dissertation. This research fills the memory management gap for the PPP execution model in particular, as well as for other parallel execution models with similar data organization and memory behavior. The contributions of this research include:

1. a dynamic memory management scheme to facilitate sharing among parallel tasks executing in a shared memory multiprocessor;

2. a flexible, low-level simulator for complete system simulation, including the parallel execution model, the cache, and the memory interconnection network of the multiprocessor;

3. a detailed simulation study of the memory behavior of a parallel execution model of Prolog on a shared memory multiprocessor architecture and the evaluation of the proposed dynamic memory management scheme; and

4. a feasibility study and a preliminary performance analysis of a two-tier memory architecture for shared memory multiprocessors which separates synchronization and write shared data from read shared and local data.

1.5 Dissertation Outline

This dissertation is divided into ten chapters:

- **Chapter 1** has provided the motivation, the research direction, and the contributions of this dissertation.
- **Chapter 2** discusses shared memory versus message passing systems and parallel execution on multiprocessors.
- **Chapter 3** provides a literature survey on memory management support for sequential and parallel execution of Prolog.
- **Chapter 4** introduces a dynamic memory management scheme for parallel execution of Prolog. This hybrid stack-heap mechanism, called *Explicitly Linked Paging Stack (ELPS)*, utilize the available address space more efficiently.
- **Chapter 5** describes the simulator used in the research. It is a complete system simulator, simulating the parallel execution model as well as the underlying multiprocessor architecture.
- **Chapter 6** presents the methodology used in validating the simulator. The simulation results are compared with those of a previously validated simulator.
- **Chapter 7** reports the memory behavior of the PPP parallel execution model for Prolog, and the simulation results of the ELPS memory management mechanism with various parameters.
- **Chapter 8** describes the Aquarius-II, a multiprocessor architecture with a two-tier memory system. This architecture is designed to reduce the synchronization bottle neck of a single bus system by using a crossbar for unsynchronized data transfers.
- **Chapter 9** reports the simulated performance results of the Aquarius-II cache and memory system.
- **Chapter 10** provides some concluding remarks and directions for future research.

Chapter 2

Multiprocessors and Parallel Execution

2.1 Multiple Processor Systems

The demand for very fast computation continues to outgrow the existing state-of-art computer systems. Steady advances in processor and memory system designs, digital circuit design, and high density packaging have greatly reduced the sequential execution time. Parallel processing holds the promise of further reducing the execution time by several orders of magnitude. Numerous architectures have been designed and built with tightly coupled multiple processors for parallel processing. With respect to the memory organization and interprocessor communication, these systems generally fall into two categories: *message-based multicomputers* and *shared memory multiprocessors.*

2.1.1 Message-Based Multicomputers

In a message-based multicomputer, each processor has access only to its private, dedicated data memory. Each primitive element is a computer (processor-memory pair). Communication among computers is either via fixed paths or via some message switching mechanism. Data sharing is done by passing messages through these specialized communication channels. In addition to the data content, a message includes a header with information regarding the source, destination, and message type. Some information in the header can be omitted if it can be deduced implicitly from the communication channel used and the

time of arrival (handshaking communication protocol). Duplication of shared data, message packing, and message unpacking add to the communication overhead.

Various communication network topology may be used in message-based systems. The Intel iPSC [Int86], the Ncube/ten [HMSe86] and the Connection Machine [Hil86] use a hypercube topology, systolic arrays [Kun82] use various array structures to fit with the intended algorithm, and transputer [Whi85] based systems can be connected in any fashion with each processor having up to 4 neighbors. Descriptions and performance evaluations of message-based systems can be found in [RF87].

Message-based architectures are scalable to thousands of processors. The Intel iPSC/d7 is a seven dimensional hypercube with 128 processor nodes. Each node contains an 8MHz Intel 80286 microprocessor, a 6MHz 80287 floating point coprocessor, 512K bytes of dynamic RAM, and 64K bytes of ROM. The Ncube/ten contains 1024 custom VLSI processors. The main reason for the scalability of message-based architectures is that the number of connections of each processor node to neighboring nodes is either constant or increases very slowly with respect to total the number of nodes. For example, each node in the hypercube has $log_2(n)$ number of connections, where n is the total number of nodes.

Efficient parallel execution on a message-based system requires that the message are small with respect to amount of work done at each processor node, and thus data sharing is kept to a minimum. Furthermore, the algorithm used must be well mapped onto the network topology, since passing a message to a distant processor incurs the latency of going through intermediate processor nodes. Programming a message-based system requires great care in problem partitioning and allocation of the processors for solving these subproblems in parallel. Parallel programming on these systems may be difficult, especially when the interconnection network has to be considered explicitly [Dem82]. How to compile for efficient execution an arbitrary program not designed specifically for a particular interconnection network is an open issue.

2.1.2 Shared Memory Multiprocessors

In shared memory multiprocessors, each processor may access any memory location. Memory may be organized in a "dance hall" fashion or distributed among the processors, as shown in *figure 2.1*. From an operating system perspective, the dance hall organization results in a *uniform memory access (UMA)*, while the distributed organization

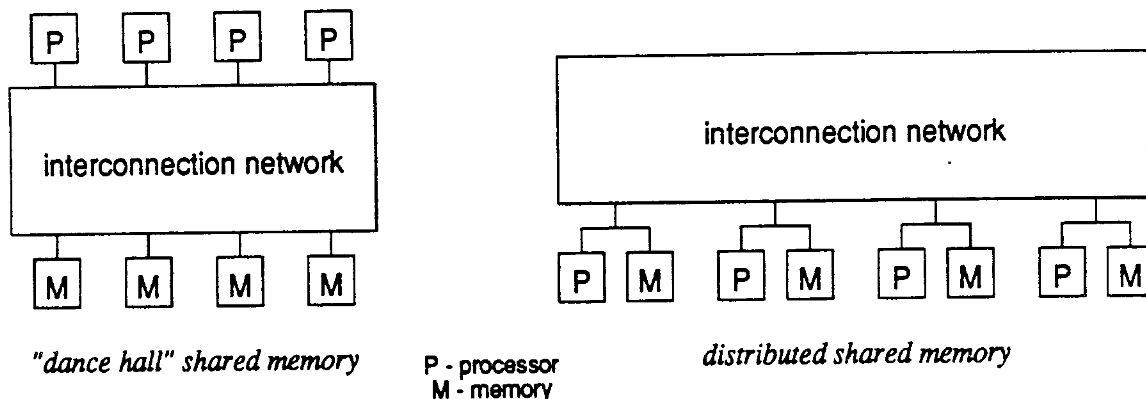


Figure 2.1: Two Categories of Shared Memory Multiprocessors

result in a *non-uniform memory access (NUMA)* [BGW89].

In the "dance hall", each processor may directly access any memory location in fairly[1] constant time. There is no local memory to each processor. Each processor may contain a local cache to speed up accesses of adjacent locations (spatial locality) and frequent accesses to the same location (temporal locality). In the context of this discussion, caches are viewed as special hardware managed buffers, and not as ordinary memory. From the processor viewpoint, variations in memory access times are due to cache misses, contention on the processor-to-memory interconnection network, and memory bank conflict. Many of today's commercial multiprocessors employ shared memory with local caches on a single bus. The Sequent Balance [TGF88], the Encore Multimax [Enc85], and Alliant FX/8 [PM86] are among the bus-based shared memory multiprocessors. These systems are generally referred to as *multis* [Bel85].

In the distributed shared memory organization, each processor has a local memory which can be accessed in fast constant time. It may also go through an interconnection network to access the memory of other processors in fairly constant time, at a much greater latency than accessing its local memory. Examples of distributed shared memory multiprocessors include Cm* [Geh87], the IBM RP3 [GF 85], and the BBN Butterfly [CGS*85]. The BBN Butterfly GP1000 can support up to 128 processor nodes, each with 4 MBytes of memory and a Processor Node Controller (PNC) that manages all memory references. A

[1]infrequent cache misses and fast fetching of a cache miss

non-local memory access across the switch takes about 5 times longer than local memory access.

Shared memory multiprocessors have a number of advantages over message-based multicomputers:

- *Efficient data sharing.* Since all memory locations are visible to all processors, shared memory systems can efficiently support extensive data sharing in parallel execution. Passing data from one processor to another requires only a pointer to where the data are stored.

- *Flexible interprocessor communication.* Interprocessor communication using shared memory is much more flexible than using messages. It can be done using software specified memory locations. Depending on the type of interconnection network used, broadcasting to multiple processors may be possible and would be much more efficient than point to point communication.

- *Ease of programming.* Proper mapping of the problem partitions onto the multiple processors is less critical than in message-based systems, and more dynamic scheduling may be done to balance the load on the processors. Given the structured topology of the processor nodes, scheduling in message-based systems is more static in nature (usually done by the programmer or by the compiler). This ease of programming also means that existing software can be compiled for parallel execution with little modifications (particularly when compiling for uniform memory access multiprocessors).

The performance of shared memory architectures depend heavily on the performance of the interconnection network. As the number of processors increases, the interconnection network becomes a bottleneck. Depending on the speed of the bus relative to the processor, the single bus can support from 4 to 32 processors before it saturates. Crossbars provide the highest bandwidth with the greatest hardware complexity (order of $p \times m$, where p is the number of processors and m is the number of memory modules). With current technology, a bit-slice 16x32 crossbar can fit on a single chip [Sri88]. Multi-stage networks are less expensive than a full crossbar, but incur a network delay in the order of $log(m)$ to go through the switch, assuming that the number of processors is less than the number of memory modules.

Compared to message-based systems, shared memory systems (particularly those with the "dance hall" organization) have two challenges to overcome:

- *Scalability.* Shared memory systems are less scalable because the number of logical connections to neighboring nodes is $n-1$, where n is the total number of processor nodes. Due to interconnection network contention, each new node can potentially block the others from accessing the shared memory.
- *Interconnection network complexity.* To reduce network contention, more complex network switches are needed and thus the network complexity and cost are increased.

However, the advantages of shared memory (particularly the ease of programming for a large class of algorithms) drive researchers to design for cost effective large scale shared memory systems. Such systems with coherent caches have been proposed for thousands of processors. These systems use a hierarchy of buses [Wil87a, Arc88, CGB89], a multidimensional array of buses [GW88, CD90], or a hierarchy of crossbars [Sri89].

2.2 Parallel Execution on Multiprocessors

Parallel processing on multiprocessors involves partitioning a problem for execution on two or more processors. This section examines the software aspect of parallel processing. The design of a parallel execution model usually includes the following two goals:

1. to provide an easy-to-program environment that requires the user to know little about the underlying architecture, and
2. to take full advantage of the multiprocessor system.

The cost of software development is a substantial part of a computer system, and often exceeds the hardware cost. The first goal provides cost effective software development and portability across different machines in the same class of architectures. The second goal exploits the performance potential of the multiprocessor architecture. The following are issues and tradeoffs involved in parallel execution:

- *Specification of parallelism.* A program may specified for parallel execution by using explicit annotations (e.g. `parbegin--parend`) or implicit parallel detection (e.g. vectorizing compilation and dataflow dependency analysis).

- *Granularity.* The length of time which a program partition runs before terminating varies from a few cycles (***fine grain*** parallelism) to millions of cycles or more (***very large grain*** parallelism). The exact size of each medium grained partition is often not known. For efficient execution, the grain size should be much larger than the creation, communication, and termination overhead.

- *Scheduling.* Compared to distributed scheduling, centralized scheduling has more information for better load balancing, but may be a bottleneck as the number of processors get to be large. The scheduler must take into account the underlying architecture, particularly the cost of task switching and task migration. The scheduler must also consider the data dependencies of the parallel tasks to quickly obtain the solution given the limited resources.

- *Data sharing.* The degree of data sharing among parallel tasks depend on the memory organization and the class of application programs. Some programs are computation intensive with few data elements while others require scanning a large database for the solution.

Up to this section, we have discussed parallel execution in quite general terms. From this point on, we will focus on the parallel execution of Prolog and its requirements for memory management.

2.3 Prolog and Its Applications

Prolog is a programming language which is based on the theoretical foundation of logic [Llo87]. Originated around 1970 from the University of Marseille, it has gained greater acceptance and popularity in recent years as a very useful language for numerous artificial intelligence, symbolic processing, and other applications. It has been used successfully for natural language processing [PS87, Dah88, HHS88], programming language compilation [VR84, CVR86], structured analysis tools [Doc88], and computer aided design for electronic circuits [BCMD87b, Clo87, Rei87, Rei88]. In addition, it has been used in a number of knowledge representation and expert systems [IH88, Shi88, WMSW87], and has been found to be very useful as a hardware description and simulation language [BCMD87a]. A logic programming language, called KL1 [KC87], has been chosen as the official language for the Fifth Generation Computer Project in Japan [FM83].

As the usage of the language increases, the demand for faster implementations of Prolog also rapidly increases. While some researchers work to optimize Prolog compilation and to devise more efficient sequential execution models, others are in search of efficient ways to exploit the great amount of potential parallelism in Prolog. Prolog naturally exhibits two types of medium grain parallelism: AND- and OR-parallelism. A number of parallel execution models have been proposed for Prolog. Some exploit only OR-parallelism [Lin84, CH86, HCH87, War87a, War87b, DLO87, BDL*88, LBD*88] or only AND-parallelism [Deg84, Her86, BR86, DeG87, Lin88, CVR88], while others exploit both types of parallelism [Con83, Bor84, Kal87, Fag87, BdKH*88, BSY88].

2.3.1 Prolog Terminology

This subsection provides a very brief introduction to Prolog, intended to familiarize "non-Prolog" readers with the language terminology, syntax, program structure, and execution semantics. This is the foundation for understanding the different types of parallelism that exist and how a parallel execution model may support them. A number of books are available on programming in Prolog [CM87, CC88, SS86]. Interested readers may consult them for a more detailed explanation of the language and programming techniques.

Prolog programs and data are represented by *terms*. Terms may be simple (variables or constants) or compound (structures). Constants are numbers or *atoms*. Atoms begin with lower case letters and variables begin with capital letters. A structure consists of a functor, which is the name of the structure (represented by an atom) and its arity, and arguments. Each of the arguments of a structure is, in turn, a term. A list is a special case of a structure with the special list functor and two arguments, the car and the cdr (as in Lisp).

A Prolog program consists of a *query* and one or more procedures (see *figure 2.2*). A procedure is defined by a set of *clauses* and it is executed by processing its clauses in sequence until one succeeds. If none of the clauses can be executed successfully, the procedure *fails*. The process by which a procedure tries successive clauses until one succeeds is called *backtracking*. It involves restoring the state of the machine to what it was before the clause was tried so that the next clause in the procedure can be tried. A clause is a complex term that consists of a *head* and optionally, a *body*. The head of a clause is also a term that has a functor (the name and arity of the functor uniquely identify the procedure

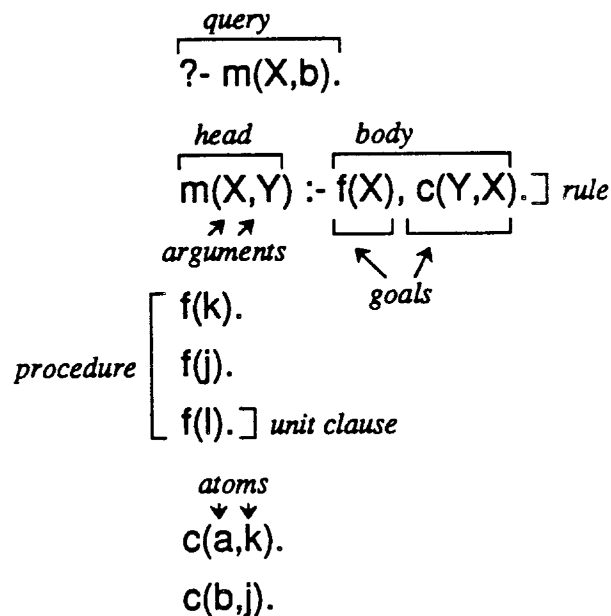


Figure 2.2: Components of a Prolog Program

of which the clause is a part) and zero or more arguments. The arguments of a clause head are also terms (simple or complex), and represent the formal parameters of the procedure. The body of a clause, if any, consists of one or more *goals* (procedure calls). Clauses that have no body are called ***unit clauses*** or *facts*; otherwise, they are referred to as ***rules***.

A clause succeeds when all the input arguments have been *unified* with the arguments of the clause head and all the goals of the clause have been successfully executed. Unification is the process by which a set of substitutions or *bindings* of the variables in the two expressions being unified result in identical expressions. If no such set of substitutions exists, the unification fails. If a goal fails, an alternate solution to the previous goal is computed. Then the goal is executed again. If no other solution to the previous goal can be found, the goal fails and an alternate solution to the goal before that is computed. Thus, Prolog finds a solution to a query by a depth first search of the solution tree. A clause fails if a consistent solution for all of its goals cannot be found.

Prolog, as a logic programming language, has the following combination of features that set it apart from other programming languages:

- *Logical, Dynamically Typed Variables.* There is a concept of *binding* for Prolog vari-

ables, in which an unbound variable gets attached to a single item. That item may be a simple value, and complex term, or even an unbound variable. The variable's type is dynamic: it changes according to the item it gets bound to. Each variable may get bound at most once, and is thus referred to as single-assignment. However, a bound variable may be unbound upon failure of the clause. From an implementor's point of view, single-assignment variables allow for a greater degree of parallelism while requiring more memory space.

- *Unification.* This pattern matching, although with a very specific set of rules, is powerful enough for numerous uses in text processing and database queries. Hardware tagging support can significantly speed up unification [KTW*86, Dob87b, ABY*87], particularly for type checking of Prolog's dynamically typed variables.

- *Backtracking.* Prolog's automatic support of non-determinism using backtracking makes it very easy to express non-deterministic algorithms in their natural forms. The cost of this support is for recording variable bindings (called *trailing*) which are undone upon backtracking. Using flow analysis, recently developed compiler techniques have made this cost insignificant [VR90].

2.4 Parallelism in Prolog

In order to adopt logic programming for parallel execution, a number of parallel logic programming languages have been introduced to avoid the backtrack mechanism that exist in standard semantic of Prolog as defined in [CM87]. These languages are referred to as *committed-choice* languages. Some examples of committed-choice languages are Concurrent Prolog [Sha86], Parlog [CG86], and Guarded Horn Clauses (GHC) [Ued85]. These languages are more suitable for operating system applications, while the standard semantic of Prolog provides a more general purpose programming language with a wide range of applications [Llo87]. Therefore, our approach concentrates on exploiting parallelism in standard Prolog.

With its simple syntax and regular structure, a Prolog program is inherently an AND/OR tree. Execution of the program is primarily a depth first, left to right traversal of the tree nodes. All the sibling AND nodes are traversed depth first, left to right, whereas an OR node is traversed only if all the siblings to the left of it had failed. Backtracking allows

for automatic exploration of previously untried alternatives. It is also the cause for a great deal of complications in efficient parallel implementation. For this reason, some researchers are looking into combining the features of Prolog and committed choice languages [HB88].

Figure 2.3 shows a Prolog program with its corresponding program tree. The arrows show the traversal of the nodes, which is equivalent to the execution of the program. The solid arrows show the forward execution, while the dashed arrows show backward execution.

The work done at each node consists of unifying the calling parameters with the head arguments of the clause, and setting up the parameters for calls to its subgoals. In addition, the work in the body of the node may involve applying some functional primitives known as built-ins for arithmetic operations, input/output, data structure manipulations, and code alterations.

2.4.1 AND-Parallelism

Inspecting the program tree, it seems natural that the branches of the tree can be executed in parallel. This has been observed and studied by Conery [Con83] and others [Deg84, Her86]. When the partitioning is done at a clause node, where calls to subgoals are to be done in parallel, it is known as *AND-parallelism*. *Figure 2.4* shows the partitioning of the tree in Figure 2.3, where the spawned processes are separated from the root process with dashed lines.

The main difficulty with AND-parallelism is the problem of binding conflict, where more than one AND subtree executing in parallel attempt to bind the same variable to different values (e.g., variable `X` in the figure above). A solution of this problem requires some synchronization mechanism for shared variables, in addition to some merging scheme for combining sets of variable bindings returned from the non-deterministic goals. Since such a scheme results in enormous run time overhead, a more constrained alternative, known as *independent (restricted) AND-parallelism*, is often chosen. This restriction requires that all subgoals to be executed in parallel must not attempt to bind a shared variable. This restriction can be fulfilled by either compile time analysis [Cha85], or by a run-time check [Deg84, Her86]. An alternative to independent AND-parallelism which does not require merging of answers is the *producer-consumers* approach [LM86, Lin88], where each variable is designated one producer goal while the other goals are designated as consumers of

```
m(X,Y) :- f(X), c(Y,X).

f(k).
f(j).
f(l).

c(a,k).
c(b,j).
```

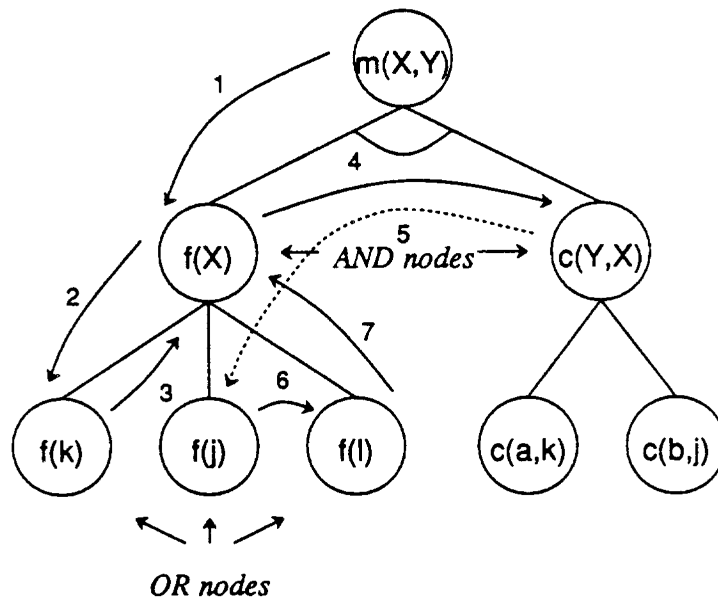


Figure 2.3: Prolog Program and Corresponding AND/OR Tree

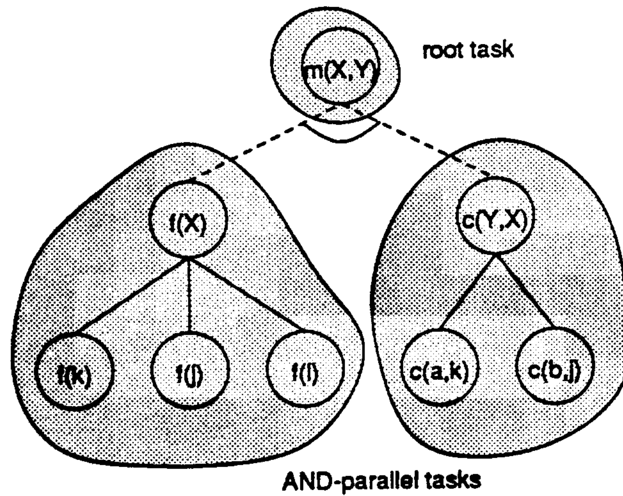


Figure 2.4: AND-Parallel Tree

the variable. A consumer goal must suspend until the producer goal of that variable has completed its execution.

2.4.2 OR-Parallelism

When the program execution is partitioned at a procedure node, with broken branches to clause nodes which show alternative clauses that give several solutions, the parallelism exploited is known as OR-parallelism. The task which executes one of the OR branches can continues with the next goal in the parent's clause. In *figure 2.5*, the task completing the first clause of `f(X)` continues with the next goal `c(Y,X)`, with `X` now instantiated to the value `k`. Thus the results are passed down the execution tree and the final solutions are available at the leaf tasks.

When OR-parallelism is combined with AND-parallelism, the results of the OR-tasks may be passed back up to the parent AND-task. In *figure 2.6*, the goals `f(X)` and `c(Y,X)` are executed in AND-tasks. OR-tasks are then spawned to execute the clauses of `f` in parallel. The results of these OR-tasks are passed back to the parent task. The OR-tasks do not proceed with the next goal (`c(Y,X)`) because it is already being executed by an AND-task. This is referred to as *containment* by Fagin [Fag87].

OR-parallel clauses may share argument variables in the head, but bindings of

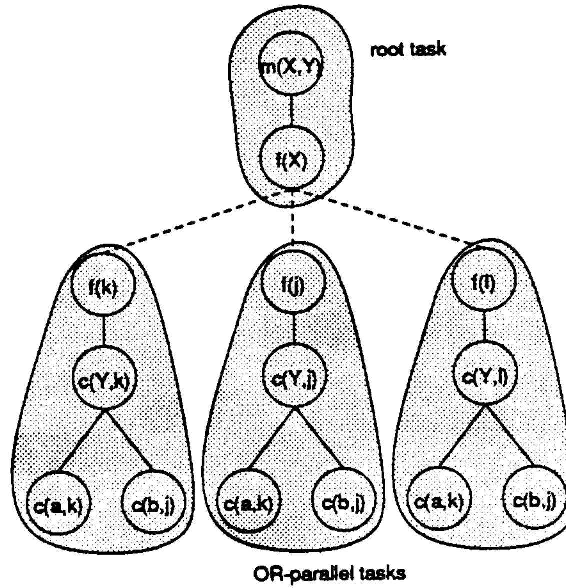


Figure 2.5: OR-Parallel Tree

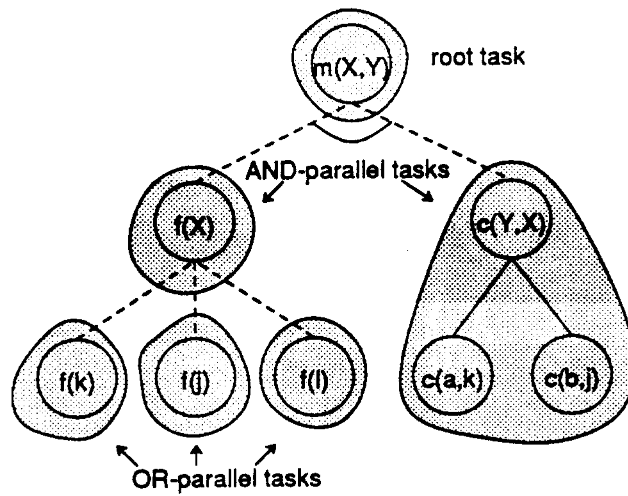


Figure 2.6: AND-OR Parallel Tree

these arguments must be hidden from the ancestor nodes until that OR node is actually traversed (in the sequential semantic order). The challenge in implementing OR-parallelism is to resolve the binding conflicts in a space and time efficient manner. For example, the OR-tasks of the goal f(X) may attempt to bind X at the same time. Thus, each of these OR subtrees must contain a separate binding environment. Efficient handling of these binding environments are being studied by various researchers [War87b, DLO87, CH86, Bor84, HCH87]. A simple simulation study by Crammond [Cra85] provides preliminary indication that the hash window scheme [Bor84] yields the best performance. However, more recently work by Warren and researchers at Argonne National Lab have presented some promising hybrid schemes combining hash windows with binding arrays [War87a]. Explanations of these schemes are provided in section 3.4.5.

2.4.3 Other Types of Parallelism

Other types of parallelism have been identified for Prolog. Consider the following example:

```
?- m(s(...), [___], X).

m(s(...), [___], X) :- a(1, X), b(X).     (m1)

a(1,X) :- ...                              (a1)
a(1,[3,5]) :- ...                          (a2)
a(2,X) :- ...                              (a3)

b([]).                                     (b1)
b([H|T]) :- ...(H), b(T).                  (b2)
```

Stream-parallelism [Sin90, LP84, Mea83] exists when a producer goal can pass a stream of values (elements of a list) to the consumer goal in a pipelined fashion. In the example above, a(1,X) is the producer of X while b(X) is the consumer. a and b can be executed in parallel, with b operating on an element in the list X while a is producing the next element.

Search-parallelism allows the heads of all clauses in a procedure to be unified with a given subgoal. This can be viewed as a simplification of OR-parallelism. In the example above, the search for the clauses that can match with `a(1,X)` can be carried on in parallel, resulting in the list of two clauses [(a1), (a2)].

Unification-parallelism [Sin90, Cit88, Sin88] carries out the unification of the arguments in the clause head in parallel. In the example above, the structure `s(...)` and the list `[_]` in clause head of `m1` can be unified with their calling arguments in the query `m` concurrently.

Depth-parallelism [Sin90, Sin88, BG87] carries out the unification of the head of a clause concurrently with the unification of a subgoal of the clause. In the example above, the unification of the arguments in `m` can be done concurrently with the unification of arguments in `a`.

All types mentioned in this subsection explore parallelism at a finer grain, below the Prolog clause/procedure level (medium grain). They will not be discussed further since this dissertation concentrates on memory management for the clause/procedure level of parallelism.

2.5 Processes and Tasks

From an operating system level perspective, a *process* is an execution environment of a program, with its own address space. This is in accordance with the definition of a Unix process [QSP85], where there is a separate virtual address map for each process. Two or more processes may share a memory block only if their virtual addresses are mapped onto the same physical page.

In this thesis, we use the term *task* to refer to a light-weight process that shares a global address space with other light-weight processes. A task contains only the execution state of the program (i.e., registers and stack pointers). Other similar terms are *thread* and *chare* [SKR88]. Tasks are spawned for concurrent exploration of the Prolog search tree. The language level notion of a task is the execution of a section of code (one or more continuous nodes in the tree), with a section of data which corresponds with that execution. Depending on the paths that tasks represent, they may be allowed to proceed in parallel, with occasional communication among each other.

2.6 Chapter Summary

In this chapter, we discuss two general categories of multiple-processor systems: *message-based multicomputers* and *shared memory multiprocessors.* We focus our attention on shared memory multiprocessors because they have the following advantages over message-based multicomputers: efficient data sharing, flexible interprocessor communication, and ease of programming.

We also present Prolog, a logic programming language that has found wide use in natural language processing, expert systems, and many other applications involving symbolic computation. We choose Prolog for our parallel execution and memory management support studies because of its intensive memory usage nature that require efficient memory management. In this dissertation, we focus on the memory management support for two types of medium grain parallelism in Prolog: AND-parallelism and OR-parallelism.

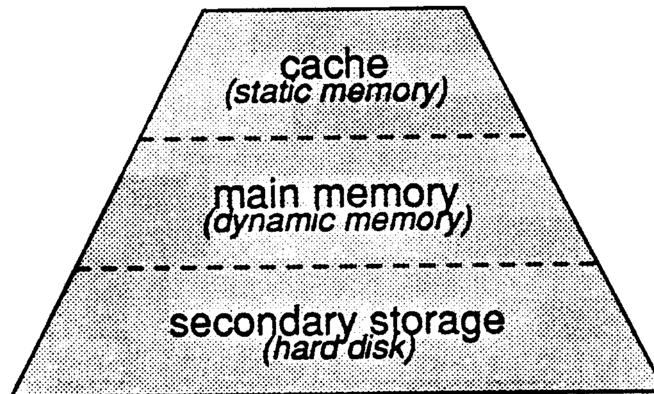
Chapter 3

Memory Management: Issues and Past Solutions

3.1 Issues in Memory Management

"Memory management" is a broad term that covers numerous issues in managing the storage space available in a computer system. The complexity of memory management increases as the layers in the memory hierarchy increases. This chapter discusses the issues in uniprocessor and multiprocessor memory management, and reviews some approaches previously taken by researchers to solve them.

Cost is a main consideration in the design of memory systems. It is often kept constant while tradeoffs are made to obtain the highest performance, namely the ability to access maximum amount of space in minimum amount of time. A memory hierarchy (*figure 3.1*) typically contains multiple layers of different types of storage devices to take advantage of the access time of fast devices (such as fast static memory) as well as the low cost, large space of slower devices (hard disks). The table in figure 3.1 shows typical speeds and costs of the various devices. The access times and prices are rough estimates based on September 1989 advertised prices. The points of interest are their access times and costs with respect to one another. At the highest level (shortest latency) of the memory, caches are kept close to processor speed. Caches are expensive because they use high speed memory chips and employ complex associative lookup. Caches also require storage space for address tags. At the other end, hard disks provide non-volatile storage and an enormous amount



Device	Speed (ns)	Cost ($)	$/Mbyte
Static Memory (32 Kbyte)	15	400	12,500
Dynamic Memory (2 Mbyte)	80	300	150
Hard disk (100 MByte)	25,000	1000	10

Figure 3.1: Memory Hierarchy and Storage Device Relative Speeds and Costs

of memory space at very low cost (and the price/space ratio continues to drop). With new technologies emerging, such as a removable 256 Mbytes optical disk for $50 (not including the optical disk drive), more space is available at much lower cost (and faster access time).

There are two key factors in memory management: *space* and *time*. Given a memory architecture with specified memory sizes and communication network structure, heuristics or algorithms are then developed to make efficient use of system resources. Regarding space, two aspects need to be considered:

1. *validity of data*

   If the data stored in a given memory location becomes invalid (or will never be used again), that space may be reclaimed; otherwise, the data must be preserved.

2. *system addressability*

   There is an upper limit to the size of memory that can be addressed. For fast access, this limit is dependent on the width of the address register and the datapath internal to the processor. Various segmentation schemes increase the addressability at the cost of loading and reloading segment registers, and limiting the address range that

can be accessed at any given time. For example, the VLSI-PLM processor [STN*88] has a 28-bit address register, but a 29-bit address space. The most significant 29th bit specifies code space or data space, and is generated by the microsequencer of the VLSI-PLM processor.

The *time* factor depends primarily on the *locality* of accesses. Cache misses, virtual memory page faults, and remote accesses to memory module via the communication network can incur severe performance penalty. Compared to a cache hit, a cache miss is typically 5 times slower, a remote memory access is 20 to 100 times slower (depending on the type of interconnection network), and a page fault is 5000 to 20,000 times slower.

3.2 Memory Management Techniques: A Historical Perspective

3.2.1 Virtual Memory

In the early days of computing, main memory was very expensive, and thus was typically small (6 to 24 kilobytes for minicomputers in the 60's [BM82] and 4 to 48 kilobytes for personal computers in the 70's [SBN82] compared to today's several to tens of megabytes for personal workstations). *Manual overlay* was a commonly used technique, where the programmer explicitly swapped a portion of data stored in memory onto disk and then swapped it back when needed. The introduction of *virtual memory* automated the disk swapping process, and freed the programmer from having to manage low level system resources. Both manual overlay and virtual memory solves the problem of insufficient space for storing valid data.

Memory is used to store input and output data, as well as intermediate results. After the data have been last used, they can be discarded and the space where they were stored may be reclaimed. Various *garbage collection* schemes scan the data space to mark the data that are still valid and to pack them together to leave the empty space for other usage. Garbage collection may be viewed as *micro* space reclamation since it operates at the single memory cell level. Deallocation of a segment of memory, described in the following section, can be viewed as *macro* space reclamation.

3.2.2 Allocation Strategies

```
1. Static allocation
        global variables in C, Pascal, Fortran
        ``static'' variables in C
        ``COMMON'' variables in Fortran
2. Explicit dynamic allocation
        via C functions ``malloc()'' and ``free()''
        via Pascal functions ``new()'', ``mark()'', and ``release()''
3. Implicit dynamic allocation
        stack: C and Pascal activation frames for function/procedure
               control information, argument data, and local variables.
        heap:  storage for Lisp and Prolog ``cons cells,'' and
               Prolog ``compound terms.''
```

Figure 3.2: Examples of Allocation Strategies

Memory may be organized into bigger chunks for allocation and deallocation. Several allocation techniques have been devised to satisfy the needs of the programming paradigms and to support the programming language features. In general, there are three allocation strategies (see examples in ***Figure 3.2***):

1. *Static allocation*
   The compiler sets aside a fixed area of memory at compile time for use at run time. Unless otherwise managed, this space can only be used for the purpose specified at compile time and is never reclaimed for other usage. The advantage of static allocation is simplicity in implementation and zero run time overhead.

2. *Explicit dynamic allocation*
   Memory management is done at run time by the programmer. In this strategy, the programmer explicitly requests a memory chunk of a specified size. Often, an allocated memory chunk can be reclaimed only at the programmer's explicit instruction. Occasionally, an "intelligent" operating system may be able to reclaim this space. This strategy provides greater flexibility, but can be quite tedious and error prone.

3. *Implicit dynamic allocation*
   Memory management is done at run time by the system, without the programmer's direct specification. This general strategy follows a stricter discipline than explicit dynamic allocation, and is thus more robust and allows more to be done automatically. The two most common structures are the *stack* and the *heap*. The *stack* structure allows space to be managed in a rigid fashion: the *last* segment that was allocated is the *first* to be deallocated. This rigidity minimize the overhead of dynamic memory management. The *heap* structure can vary greatly in implementation, but it is always much less restrictive than the stack structure (allocation and deallocation can be in arbitrary order). A common characteristic in heaps is some sort of bookkeeping of the dynamic space usage, either explicitly with special memory management pointers to allocated areas and/or free areas (e.g., a free list), or implicitly by tracing through active data objects. This characteristic allows space to be partially or fully reclaimed.

3.2.3 Data Organization and Memory Access Policies

Average memory access time can be reduced by organizing data to increase *locality*. The stack structure exhibits greater locality than the heap structure. Tradeoff decisions can also be made on whether to *copy* or to *share* data. Copying data is advantageous if the cost of copying is offset by the time savings for faster accesses to the local copy of the data. Throughput can be improved by overlapping operations. For example, a process may be swapped out while waiting for a page fault to be serviced.

3.3 Sequential Execution of Prolog

3.3.1 Understanding Prolog Memory Requirements

For memory management to be more "intelligent," more information is needed regarding the size of the data objects and the type of accesses that these data objects may have. This section examines specifically the storage requirements of a Prolog engine.

As in any language, Prolog has three different entities that require storage: code, data, and control information. As shown by figure 2.2, a Prolog program consists of a query and a set of *procedures*, with each procedure containing one or more *clauses*. Prolog procedures have two faces. In some cases, they are executable code. In other cases, they act as

data stored in a database, with the procedure name (or *functor*) as the index. In the most complex case, data is manipulated and transformed into executable code. `Assert()` and `retract()` are Prolog builtin procedures that can modify the code database. Assert and retract present great implementation complexity. They involve symbol table management for on-the-fly compilation, efficiency of compilation and of code produced by on-the-fly compilation, and proper linking with existing code for desired semantics. In this dissertation, we concentrate on the issue of memory management for data and control information, and do not handle assert and retract.[1] We treat Prolog code as a static entity which cannot be modified at run time. Thus a segment of memory can be statically allocated for storing code.

There are four major types of data objects in Prolog: variable, atom, list, and structure. Variables and atoms are of fixed length, usually require only one memory cell each (the cell for an atom contains a pointer to the string table). Lists and structures, on the other hand, can be arbitrarily large and are very dynamic in nature. These data objects must be retained as long as the program does not backtrack. Thus, a heap is deemed appropriate for providing dynamic storage for Prolog data objects.

Two types of control structures are needed for Prolog. First, an *activation frame* is needed to store the return address at each procedure invocation. The pointer to the previous activation frame and local procedure variables are also stored in the activation frame. The *backtracking* feature of Prolog requires an *alternate clause frame* to store information regarding the next alternate clause to execute in case the current clause fails. The alternate clause frames behave in a last in first out manner, with the last alternate clause frame containing the first alternate clause to execute in case of failure. Thus, a stack is the most appropriate mechanism for storing alternate clause frames.

With a better understanding of storage requirements of Prolog, *how can we best manage memory for sequential execution of Prolog?* The following section presents one very well known answer.

[1] In place of assert/retract, we provide two Prolog builtins **set/2** and **access/2**, which can behave like assert and retract for a restricted class of Prolog procedures: the procedures with exactly one unit clause each. Many Prolog programs which use assert and retract to store into and retrieve from the database can be easily modified to use set and access instead.

3.3.2 The WAM Stack Model

The Warren Abstract Machine (WAM) [War83] is an efficient engine for sequential execution of compiled Prolog code. It is the single most widely studied abstract machine for Prolog. The WAM has been simulated at the instruction level and register transfer level [Dob87b] and implemented in microcode on the general purpose machines such as the NCR/32-000 [FPSD85] and the VAX/8600 [JMP87]. Extensive performance studies have been conducted by many research teams [Dob87b, Tic87, TD87]. The Programmed Logic Machine (PLM) [Dob87b] is a specialized architecture developed at the University of California at Berkeley to support an extended WAM instruction set. The PLM has been built in TTL [DPD84] and VLSI [STN*88]. Other WAM-based architectures include the Xenologic X-1 [Dob87a] (which is the commercial successor of the PLM), the Knowledge Crunching Machine (KCM) [BBB*89], and PSI-II [NN87].

One of the most notable features of the WAM is its stack model for data storage, which allows for efficient space recovery upon backtracking. The WAM memory model consists of four stacks: the local stack, the global stack, the trail stack, and the push-down list (PDL). The *PDL* is a small stack used for unification of nested lists and structures. Its use is only within a unify instruction and thus will not be included in the discussion of the major stacks. The PDL is on-chip in the VLSI-PLM [STN*88], the VLSI implementation of the PLM architecture. The other three major stacks together form a set of stacks, called a *stack set* [Her86].

The *local stack* contains the two types of frames described in the previous section. Alternate clause frames, called *choicepoints*, save the state of execution (argument registers and stack pointers) before trying one of several candidate clauses of a procedure. If the clause fails, the choicepoint is used to restore these registers and stack pointers before the next clause is executed. Activation frames, called *environments*, are similar to call frames in a traditional programming language, with storage allocated for local variables and procedure return pointer. Unlike traditional call frames, however, environments in Prolog cannot always be deallocated after the procedure succeeds since the procedure may be reinvoked to produce another solution. If a procedure has alternatives, there will be a choicepoint above the environment of the procedure call.

The *global stack* is used to store dynamically allocated data structures (Prolog variables, atoms, lists and structures) built up by the program. In the PLM, the global

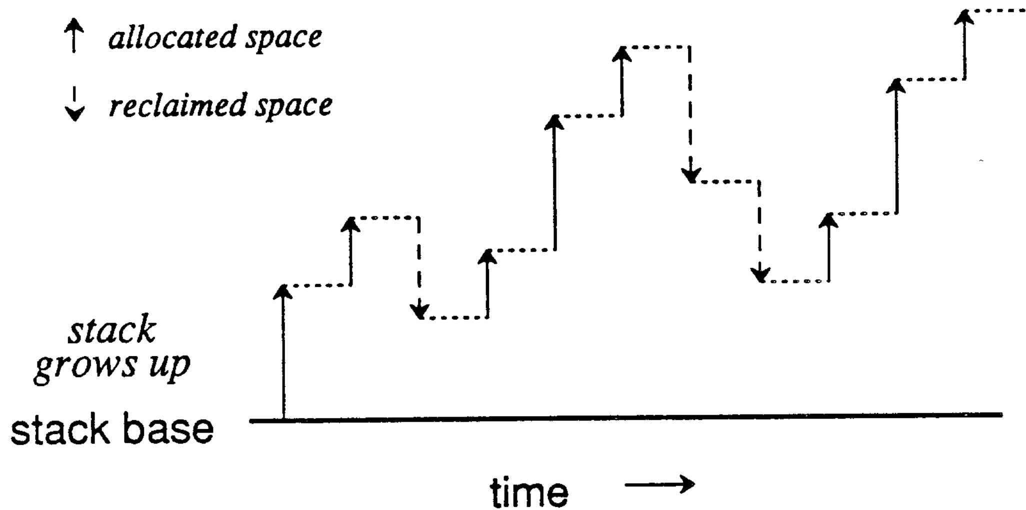


Figure 3.3: Stack Usage Behavior

Each up arrow represents the amount of space requested during execution. The sum of all up arrows is the total "allocated space." Each down arrow indicates the amount of space reclaimed upon backtracking. With the stack mechanism, much of the allocated space can be recovered and reallocated for new requests.

stack is called the *heap* because access to data stored in the global stack is not in a strict last-in-first-out (LIFO) manner. The heap backtrack (HB) pointer marks the top of the global stack at the time that a choicepoint is allocated. Upon failure of a clause, the current top of the global stack pointer can be reset to the HB pointer, thus reclaiming all the space that was used by the failed clause. *The backtracking feature of Prolog allows this special reclamation of space of a heap-like structure in a stack-like fashion.*

The *trail* is used for storing bindings made during execution of a clause. When that clause fails, all variables bound in that clause are reset to unbound, and the space used to trail these bindings are recovered when the top of trail pointer is moved back to where it was before the execution of the clause.

The WAM's multiple stack mechanism is also present in several other Prolog engines [Yok84, Clo85, KTW*86]. For non-deterministic programs which perform extensive backtracking, the WAM stack mechanism is extremely efficient, allocating and deallocating space at minimal cost. A study by Touati and Hama [TH88] indicated that for some pro-

grams, the maximum stack space used is less than 10% of the total allocated space [TH88] (*figure 3.3*). Thus, over 90% is automatically reclaimed by the stack mechanism upon backtracking, without the need for garbage collection. For deterministic programs which rarely backtrack, this automatic space reclamation is much less, but is still significant. In the best case, the stack grows to a maximum size of 48% of the allocated space. (In the worst case, the stack grows to maximum size of 100% of the allocated space, thus requiring garbage collection.)

Within this stack mechanism for sequential execution of Prolog, several tradeoffs can be made that affect memory space usage. One such tradeoff is *structure sharing* versus *structure copying*. Under structure sharing, the compound term (a Prolog *structure*), is represented as a pair of pointers (pointing to the code of the structure and an instance of the binding environment) which are used to access structure value. Variables with same structure value would contain the same pointers, thus sharing that structure value. Structure copying makes a duplicate copy of the structure for each instance of the variable. Structure sharing allows fast building of compound terms, but requires more time to access them. Discussions of structure sharing versus structure copying and other details in memory management for sequential implementations of Prolog can be found in [Mel82] and [Bru82].

3.4 Parallel Execution of Prolog

3.4.1 Memory Requirements

Prolog tasks (described in *section 2.4*) are potentially numerous since each task is used for the traversal of a small section of the execution tree. The number of tasks is inversely proportional to the granularity (or size) of each task. This granularity is difficult to control statically and may be expensive to control dynamically. The tasks may also take up a widely varying amount of memory space. Figures 2.4, 2.5, and 2.6 demonstrate this potential variance. In figure 2.6, the AND-tasks and OR-tasks are of different sizes (in terms of number of nodes in the tree).

Because of the support for non-determinism in Prolog, a task needs to retain its state for future backtracking. It also needs to preserve its data space if the variable bindings are passed by pointers to the data space and not by copying. This may result in a very large number of tasks which may never be executed again. Depending on how the data space for

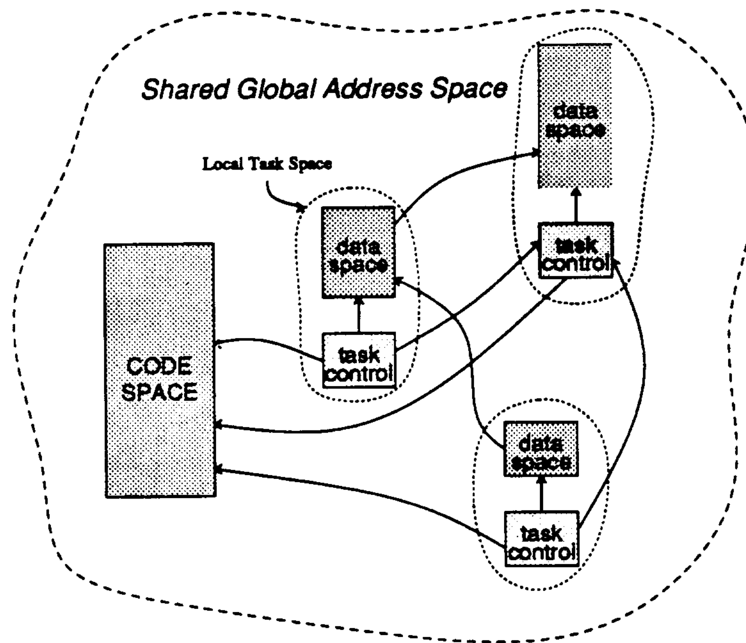


Figure 3.4: Tasks in Global Memory Space

the task is allocated, some address space could be tied up unnecessarily when it could be used for spawning new tasks or for running existing tasks.

Figure 3.4 gives a conceptual view of tasks existing in memory, with each task having a control block and an associated data space. To avoid the high overhead of copying, the tasks may share access to each other's task space via passed pointers. A *global address space* is needed to efficiently implement this extensive sharing. The question at hand is: *How can the address space be adequately distributed among the tasks and globally managed for efficient parallel execution of Prolog?*

We are interested in obtaining a *single-solution* for a Prolog program, where the OR nodes of the execution tree are partially traversed to obtain only one solution. This is in contrast to *all-solutions*, where all nodes in the execution tree are always explored for all possible answers. The memory management scheme should effectively support this single-solution objective.

3.4.2 Scheduling Effects on Memory Behavior

Task scheduling is a very important factor in the resulting memory behavior. The scheduler, either centralized or distributed, affects memory behavior in two ways:

- **task execution order.** The scheduler determines which task gets to execute first. Since each task represents a different part of the overall execution tree, the order of execution affects the *spatial* locality of the dataspaces of the tasks. From space reclamation viewpoint, it is desirable to have close together the dataspaces of AND-tasks in closely related subtrees. From a parallel execution viewpoint, these dataspaces should be far enough apart to not introduce any extraneous memory contention (different cache block, different memory module).
- **processor assignment.** The scheduler decides which processor will execute a task. From a cache performance viewpoint, tasks that share data should be executed in the same processor. From parallel execution viewpoint, tasks that can execute in parallel should be assigned to different processors.

Another issue with scheduling Prolog tasks is the management of potentially useless work. With AND-parallelism only, work done by sibling AND-tasks to the right of an AND-task is potentially useless if that AND-task fails. With single solution OR-parallelism, many OR-tasks may do useless work if their results do not affect the final solution. And yet, these OR-tasks take up processing power as well as storage space. A good scheduler should minimize the amount of useless work while maximizing the amount of parallelism.

Numerous parallel execution models have been proposed for Prolog [Con83, Her86, Kal87, War87b, BSY88]. Some have been simulated while others have been implemented on multiprocessors. From a memory management viewpoint, these models can be classified into two categories: models for *message based* (non-shared memory) multicomputers and models for *shared memory* multiprocessors. The next two sections will review a number of memory management schemes previously proposed. The focus of this thesis is on shared memory multiprocessors. Message based models are briefly covered for the sake of completeness.

3.4.3 Message-Based Models

A number of models for parallel execution of Prolog have been proposed for message-based multicomputers. This section briefly reviews the memory management

scheme for each of these execution models.

"Closed Environment" in OPAL

The *closed environment* model [Con87] was developed for OR-parallelism in OPAL (Oregon PArallel Logic), which is an implementation of the *AND/OR Process Model*[Con83]. In this scheme, an activation frame for a clause is extended to contain a copy of the unbound variables from the ancestor nodes. The parent frame is copied into the child process, and a *closing* algorithm is applied to a frame to remove all links from a frame to ancestor frames. All references can be resolved within the two frames used in unification.

"Closed Tuple" in the Reduce-OR Model

The *Reduce-OR* model [Kal87] performs a *tuple closing* to remove all references outside of a frame. Unbound variables are duplicated, while ground terms may be shared. Details of the two-phase unification algorithm is provided in [KRS88]. There are similarities between this scheme and the *closed environment* scheme described above. A major difference is that this memory model is designed for both AND- and OR-parallelism while the closed environment model as described in [Con87] is suitable for OR-parallelism only. The Reduce-OR model has been implemented on various shared memory multiprocessors (Alliant FX/8, Encore Multimax, and Sequent Balance) and a message-based machine (Intel iPSC/2 hypercube) [SKR88]. In the shared memory systems, optimizations are made to reduce the amount of copying.

Local Bindings in the Limited-OR/Restricted-AND Model

The *Limited-OR/Restricted-AND Parallelism (LORAP)* [BSY88] is another model for both AND- and OR-parallelism designed for a distributed memory system. An emulation of the LORAP abstract machine has been implemented on a network of Transputers with a wrap-around mesh topology. The LORAP-abstract machine contains multiple processors, each with its own local memory. Interprocessor communication is done by passing messages over dedicated links between two processors. A number of processes are statically created for each processor. A cell is created in the child process for each unbound variable in the parent process, with links to the parent variable. A child may directly bind its own copy of the variable. When passing back the results, the values of the bound variables in the child process are unified into the parent process.

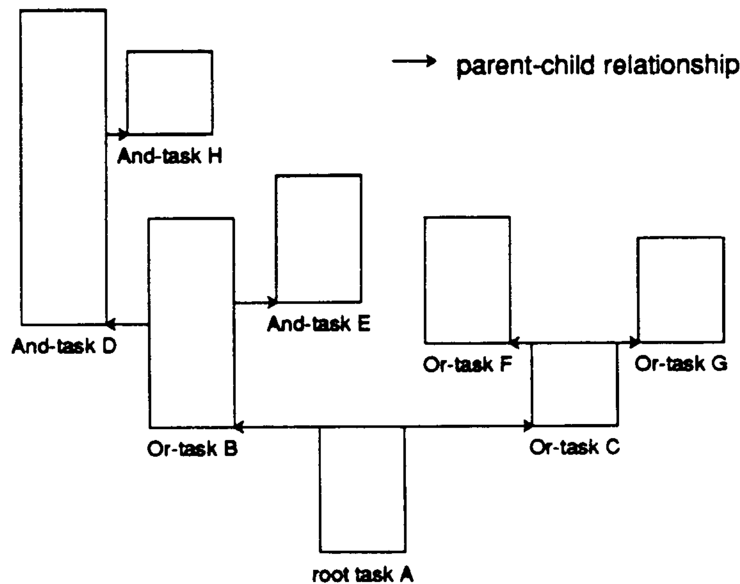


Figure 3.5: Conceptual View of a Cactus Stack

3.4.4 Shared Memory Models

As discussed in section 3.3.2, the stack is an efficient mechanism for reclaiming memory upon backtracking. Many parallel models use an extended stack mechanism, called the *cactus stack*, for parallel execution of Prolog. Figure 3.5 shows a conceptual view of the cactus stack, which is a tree structure with a stack at each node. Execution begins with the *root* stack. A new stack is created for each task spawned and is branched out from the current stack. Depending on the model, execution on the parent stack can either suspend or continue in parallel with execution on the child stack.

Implementation of a stack requires a segment of memory, a stack base pointer, and a stack top pointer. In actual implementation, each branch of the conceptual cactus stack may be an independent stack or several branches may share the same stack, since all branches are not active during the same period of time. The WAM-based parallel execution models designed for shared memory systems use three general types of memory models: a stack set for *each processor*, a stack set for *each task*, and a stack set for *one or more tasks*. In the first type, multiple tasks are allowed to share the same stack set, interleaving the data space of each task onto the same stack set (as shown in *figure 3.6*). Execution models that

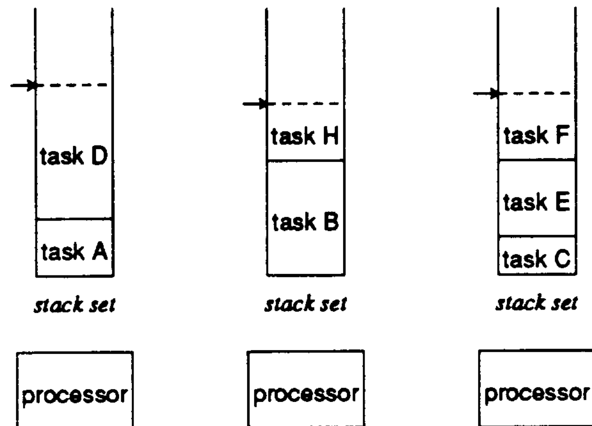


Figure 3.6: Stack Set per Processor Memory Model

use this memory model include RAP-WAM [Her86] and APEX [Lin88] for AND-parallelism, and the SRI Model [War87b] and Aurora [LBD*88] for OR-parallelism. Aurora has been implemented on both a single-bus shared memory system (Sequent Balance) [Sze89] and a distributed shared memory system (Butterfly GP1000) [Mud89].

To prevent leaving holes[2] in a stack and to ensure that the space of the executing task is at the top of the stack, various ordering schemes are used in the scheduling of the tasks for parallel execution at the cost of some restriction on parallelism (e.g., the various *steal rules* described in [Bor84, Her86, Lin88]).

In a stack set for each task, the task space is independent of the processors (shown in *figure 3.7*), thus allowing for more flexible scheduling and a higher degree of parallelism. For example, the PPP [Fag87] assign one stack set to each AND-task and each OR-task. The entire stack set can be discarded when a task terminates.

The third type of memory model is a relaxation of the first type, allowing for more stack sets than the number of processors. This relaxation has been shown to increase the degree of parallelism, resulting in faster execution in APEX [Lin88]. Borgwardt's model [Bor84] for AND-, OR-, and stream parallelism also falls under this category, allowing the AND-tasks to share the stack set, while creating a new stack for each OR-task.

[2] A hole is the space previously occupied by a task that has terminated but cannot be reclaimed since it lies below the space of an active task.

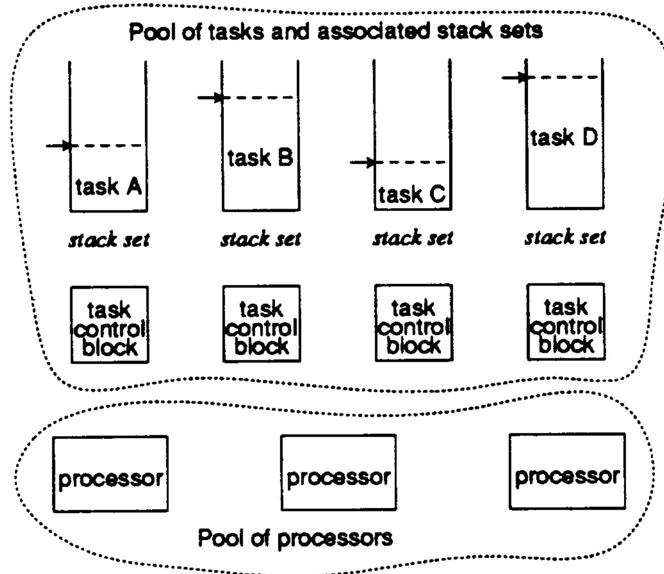


Figure 3.7: Stack Set per Task Memory Model

3.4.5 Or-Parallel Binding Environments

Or-parallelism introduces a special memory requirement. Sibling tasks that traverse alternative OR-branches in parallel (called *or-tasks* [Fag87]) may attempt to bind the same variable to different values, with one of them represent the current result and the others represent alternative solutions. This binding must be stored such that it is not visible to sibling or-tasks, and is visible only to the parent's task when it is requested. In the message-based models (section 3.4.3), each task makes the bindings in its own local space and the results are copied back to the parent task when needed. In the shared memory models, various schemes have been proposed to solve the Or-parallel binding problem. These schemes are discussed below. This section is intended to give the reader an overview of the many possibilities. This covers the comparative studies by Crammond [Cra85], Ciepielewski and Hausman [CH86] and Warren [War87a].

Binding List with Time Stamps

In Tinker and Lindstrom's [TL87] implementation of an Or-parallel Prolog on the BBN Butterfly^{TM}, there is a binding heap in each processor. *Time stamps* are used for

choicepoints (to indicate the relative time of choicepoint creation) and for value cells (to indicate relative time of binding), and are issued locally on each processor. Bindings for the same variable are made in each processor's binding heap, but are linked together to form a binding list. There is an "ancestor stack" on each processor to keep a history of computation (e.g., processor 1 is an ancestor of processor 2), used to determine the appropriate value in the binding list.

Advantages: Because time stamps are used to determine the appropriate binding for the requesting processor, no unwinding upon failure is needed and thus no trail is kept for the bindings. Task migration incurs low to medium overhead. Only the ancestor stack and the information in the choicepoint need to be copied to the new processor.

Disadvantages: Time stamps cost extra memory and time (but have uses in intelligent backtracking and tracing for debugging) [MU86]. Dereferencing requires traversing the binding list, which can be expensive. It is bounded by the smaller of the number of processors and the number of bindings. Major drawback of the linked-list method is that value cells on the binding heap that become unused cannot be recovered.

Hash Windows

Borgwardt's scheme [Bor84] creates a hash table (or window) in the global stack of each Or-task. Each entry in the window contains an [address, value] pair. When a task binds an unbound variable inherited from its parent, the value is stored in the hash window at the hash address of the unbound variable in the parent's stack. Hash tables of parent and child are linked into a linear list. When dereferencing, a task first looks into its own hash table. If not found, it recursively searches up the chain to look in the parent's hash window. Other execution model that also use hash windows include the PPP [Fag87], the Argonne Model [SW87, War87a], and PEPSys [BdKH*88].

Advantages: Since hash chain contains only descendants of one another while the binding list may contain bindings of siblings. Ancestor relationship is implicit in the hash chain, no separate ancestor stack is needed as in binding list.

Disadvantages: There is extra complexity involved in handling hash collisions and hash window overflow. Furthermore, the length of the hash window chain is unbounded. To reduce the access time, the Argonne Model has the concept of a "favored" binding, where each shared variable is associated with a processor. This processor may bind the variable in place with a special bit to indicate that the binding is only relevant to the favored processor.

Unfavored bindings are stored in the hash window. The favored bindings may be accessed in fast constant time. The Manchester-Argonne Model [War87a] proposes the merging of parent and child hash windows when all except one Or-paths have been explored.

Variations: The hash window scheme contains many variations. First, all unbound variables could be copied into the child's hash window, making the first dereference very fast. Second, if a bound variable is found in the parent's hash window, it could be duplicated in the child's hash window to save time on subsequent searches. Both of these techniques increases the size of the hash window, and the amount of time saved depends on the access frequency of the variable by that Or-task.

Variable Importation

In Lindstrom's *variable importation scheme* [Lin84], each task contains a local variables vector. Unbound variables from the parent task are imported into the child's task via an *import mapping vector* equal in size to the parent's variables vector, and additional slots are created in the local variables vector to store bindings for the imported variables. Upon termination of the child task, a new variables vector is created for the parent task, with the previously imported unbound variables updated with new bindings and new slots created for unbound variables in the child task. An *export mapping vector* is used to export unbound variables from the child task to the parent's task.

Advantages: Dereferencing requires at most a two level search, in the local task or the parent's task. Thus, memory accesses have high locality of reference, especially since memory writes are done only on the local task.

Disadvantages: The algorithms to import and export variables are complex and can be expensive.

Directory Tree

In the basic model of the *directory tree* proposed by Ciepielewski, Haridi and Hausman [CH83, CH86], the binding environment of each task contains a directory pointing to a set of *contexts* (activation records) containing the values. The address of each variable is a triple: [directory address, context offset, variable offset]. When a clause is invoked, a new context is created and its address is placed in the task directory. When a child task is spawned, a new directory is created. The parent's directory is scanned. References

to contexts with no unbound variables (committed contexts) are duplicated in the child's directory, thus allowing the child to share the parent's contexts. Contexts with unbound variables (uncommitted contexts) are copied into the child's local space.

Variations: There are a number of variations to the basic model. They are as follows:

- *Delayed context copying.* The copying of uncommitted contexts can be delayed until the context is first accessed. This avoids copying contexts that are never used.
- *Copy on read.* When a (committed or uncommitted) context is read, it can be copied into the local task to increase locality for future accesses, incurring the extra cost of copying committed context.
- *Directory tree.* The copying of the entries in the parent's directory into the child's directory is delayed. When first created, the child's directory contains pointer to the parent's directory. Using the ***local context strategy***, the child's directory also gets the reference of the most recently created context in the parent's directory.
- *Hashing on contexts.* The directories are of a fixed size and a hash function is used to enter context references into the directories.
- *Hashing on variables.* In this scheme, the values are stored directly in the directories, as in Borgwardt's hash window scheme (previously described).

Ciepielewski and Hausman simulated the various combinations of the variations described above for both a dance hall and a distributed shared memory multiprocessors. The general conclusion drawn by them is that [CH86, page 254]:

> "the straightforward implementation [with delayed copy and no copy on read] is good when the search tree is shallow (cheap process creation), distance between branching point is large (the same directory is used under several unifications), many variables in the same context are used (smaller copying overhead per variable), and finally when contexts are small (small copying overhead)."

When the opposite conditions hold true, hashing on variables performs better than the other variations.

Versions Vector

Hausman, Ciepelewski, and Haridi also proposed another storage model for Or-parallel execution, called the *Versions Vector* model [HCH87]. In this model, a vector of size equal to the number of processors is created for each shared variable. To bind a shared variable, the processor puts the value in the appropriate slot of the versions vector and enter the variable address into the trail stack. In one aspect, this scheme may be viewed as the binding list of Tinker and Lindstrom's model combined into a vector form. However, the versions vector WAM uses the trail instead of time stamps.

This scheme allows fast, constant time look up but requires more storage space and expensive task switching. When a processor switches to a different path of the search tree, variables in the old path needs to be untrailed and variables on the new path need to be installed. The paths are the segments from the old node and the new node up to a common ancestor node. Two optimizations are made to reduce this task switch cost. First, *promotion* copies the bindings in some version vectors to the original stack position. Subsequent dereferences will find the variable bound in place. And second, *delayed installation* postpones the installation of variables in the new path until they are accessed.

Binding Array

The *Binding Array* model, independently proposed by D.S. Warren [War84] and D.H.D. Warren [War87b] [3], allows fast access time at the cost of slower task switching time. In this scheme, each task contains a binding list (called *forward list* in [War84]). When a variable in an ancestor's task is bound, the [address, value] pair is entered into the binding list. Each processor contains a buffer, called the *binding array*, which contains all shared variables (bound and unbound) along the path on which the processor is exploring. The binding array is updated when new bindings are made or when the processor task switches to a different path in the tree.

This model allows fast access to the variables at the cost of task switch time. Whereas the Versions Vector model keeps one versions vector for each variable, this models keeps one binding array of several variables for each processor. Thus this scheme has greater locality and does not have the potential synchronization conflict which exists when several processors attempt to bind the same variable. As in the Versions Vector model, the task switch overhead can be reduced by keeping the task switching to adjacent nodes.

[3] D.H.D. Warren reportedly formulated the binding array concept while at SRI in 1983, but did not publish until much later.

3.5 Static Partitioning of Globally Shared Space

In the shared memory models described in section 3.4.4, there are a number of stacks for each task or each group of tasks. A straightforward method of managing the globally shared address space is static partitioning. The globally shared address space is divided into equal partitions, one for each stack set. The space for each stack set is further subdivided into space for each of the stacks. Thus the space of each stack is inversely proportional to the number of stack sets. When this number is small, the space allocated to each stack is sufficiently large that overflow would rarely occur. When the number of stack sets is very large, the stack space is very small and thus the chance of stack overflow is greatly increased. This is particularly true of the stack-set-per-task models, such as the PPP Execution Model [Fag87]. In the PPP execution model, tens of thousands of tasks may be spawned over the life time of the program to exploit the medium grain parallelism. Many of these tasks will terminate, giving up their spaces for future tasks. However, many others will go into the sleeping state, holding on to their execution state and data spaces for potential future backtracking. These sleeping tasks accumulate over time, tying up statically assigned but unused memory space that could be used to spawn new tasks. The PPP execution model takes the simple approach of reverting to sequential execution when no more space is available for spawning new tasks.

3.6 Solving the Problems of Static Partitioning

Three well known techniques may be used individually or together to reduce the problems of static partitioning. While they are very useful for some situations, they each have shortcomings of their own. This section discusses their advantages and disadvantages.

3.6.1 Virtual Memory

Extending the virtual memory space does not solve the problems of static partitioning, although it can reduce the chance of overflow. The globally shared address space can be extended up to the width of processor's memory address register and width of the internal processor datapath (it would be too inefficient to require multiple cycles to transfer pointer data). Mappings of virtual to real addresses are kept in page tables, and hardware support is needed for address translations and for page table caching. Virtual memory in-

creases the complexity of processor, cache, and bus design. Larger virtual addresses require a wider address bus (for virtual addressed caches) and greater bus bandwidth (for pointer data), and this extra cost and complexity does contribute to performance. The virtual address space, which is typically 32-bit in present technology, may even be insufficient when it is divided for a very large number of tasks. Overflow may still occur in one partition when its statically allocated space is exceeded.

Virtual memory is best for the cases where the actually used space is larger than the available physical memory (and disk space is used to fill the gap), where access to certain address spaces is restricted, or where usage of the virtual address space is very sparse. The memory management problem described in this paper is quite different. The space allocated for a stack does not actually get used until data is pushed onto the stack. Much of the address space statically allocated to a task remains unused, but unavailable for use by other tasks.

Segmentation techniques that use a segment register (or a segment table) to extend the global address space do not extend the *shared* address space, since not all segments are accessible at any given time, without reloading the segment base registers. For example, the SPUR multiprocessor system [HEL*86] has a 32-bit *process virtual address (PVA)* and a 40-bit *global virtual address*. The top two bits of the PVA are used as an index into a 4-entry segment table. Thus at any given time a process may access at most 4 segments.

3.6.2 Garbage Collection

Space containing inactive data which are no longer accessible may be reclaimed by garbage collection. Various *mark and compact* techniques have been effectively used to garbage collect the global stack of the WAM [PBW85, ACHS88, TH88]. However, parallel garbage collection is very difficult to perform efficiently. Garbage collectors for parallel systems generally fall into two categories [Zor89]: *on-the-fly* and *stop-and-copy*. On-the-fly garbage collection algorithms allow collector processes and user processes to execute concurrently, while stop-and-copy algorithms suspend all processors during collection. The disadvantage of on-the-fly collection is that collector and user processes must be carefully synchronized for correct execution. In the simplest but slowest stop-and-copy scheme, garbage collection is performed by only one processor. In the faster schemes, garbage collection is performed by several processors running in parallel. In any case, garbage collection

does not completely solve the space allocation problem. If the statically partitioned task space containing valid data is exceeded, it can not be further compacted. Furthermore, if a task terminates in the near future, its entire space can be quickly discarded. In this case, the time spent to garbage collect before the task terminates can be saved by obtaining new free space and delaying garbage collection until no more free space is available.

3.6.3 Copy When Overflow

When a stack overflows, a larger area of free space may be used to copy over the old stack. The complexity and cost of this operation are similar to those of parallel garbage collection, since the pointer data must be updated to point to new addresses. Furthermore, copy-when-overflow does not deal with the underflow problem. When the stack usage shrinks, the unused space on top of the stack remains allocated to that stack.

3.6.4 Dynamic Allocation

Dynamic allocation may be used in place of static partitioning to adapt to the changing memory requirements of the tasks. In some cases, it may stand alone as the memory management technique for parallel execution. It may also be integrated with one or more of the three techniques described above for more complete memory management. In the next chapter, we will describe a scheme for dynamic allocation and deallocation which allows for more efficient sharing of the global address space.

3.7 Chapter Summary

In general, memory management has two aspects: space and time. Sufficient space must be allocated to where it is needed and reclaimed when it becomes unused. The allocation, deallocation, and data access times should also be minimized. For sequential execution of Prolog, the stack structure is an efficient memory management mechanism because space can be quickly reclaimed upon backtracking. For parallel execution, the *cactus stack* is a conceptual memory model that allows parallel tasks to share data with their ancestors. Each branch of the cactus stack is the local data space of a task. (A global address space can efficiently support the sharing of data among the medium grain, parallel tasks.)

In actual implementation, tasks executing in the same processor may share a common stack, or each task may have its own stack. The first scheme restricts parallelism for more efficient reclaiming of space, while the second scheme has a serious problem with space allocation. As the number of stacks increase, the stacks are much more likely to overflow as the shared space is partitioned into smaller segments (under static partitioning). Various techniques may be used to reduce this problem. They include: virtual memory, segmentation, garbage collection, and copy when overflow. The advantages and disadvantages of these techniques were discussed in section 3.5. In the next chapter, we will present a dynamic allocation scheme that allows for efficient sharing of the global address space.

Chapter 4

ELPS: The Explicitly Linked Paging Stack

4.1 General Model

The schemes previously described maintain a contiguous address space for each stack. An alternative would be to allocate address space in small ranges as needed, and linking these segments to form a conceptual stack. This chapter presents such a scheme, called *ELPS (Explicitly Linked Paging Stack)*, which is basically a heap management mechanism adapted to provide dynamically sized stacks.

The concept of linked segments of memory is a classic one. Operating systems manage pools of free pages to be allocated to user processes. Support libraries for the C programming language contain memory allocation/deallocation functions for storage of dynamic data. One important distinction is that in these memory management support, allocation and deallocation of space must be explicitly requested by the programmer. ELPS provides automatic (implicit) memory management support for stacks in a parallel execution environment.

4.1.1 Page Partitioning

In ELPS, the globally shared address space is divided into many small (thousands of words) chunks of equal size, called *pages*. Since it is difficult to determine at compile time (or task creation time) how much space a task will need, equal sized chunks are

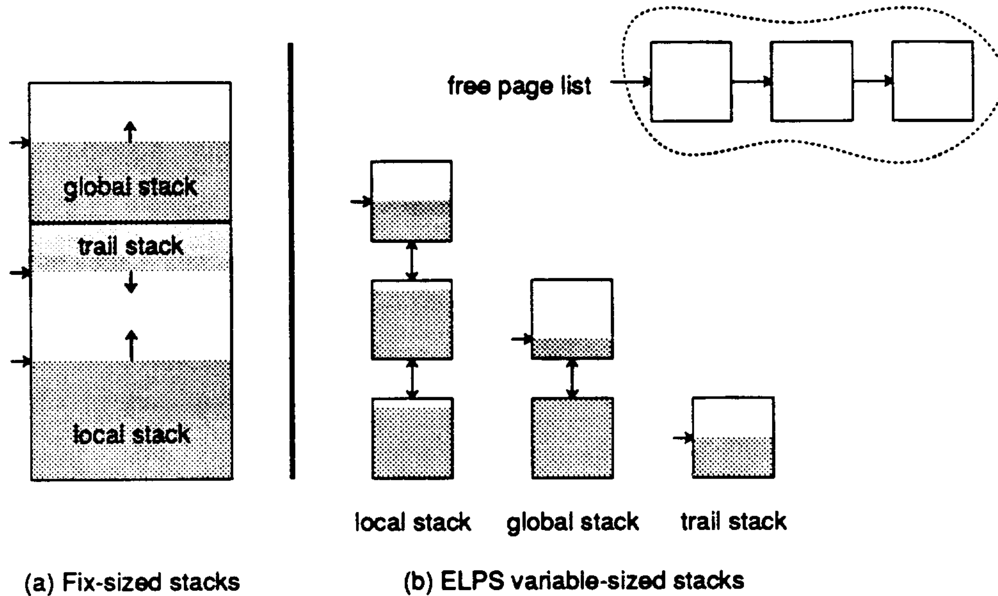


Figure 4.1: Fix-sized versus ELPS Variable-sized Stacks

adequate and require less bookkeeping than variable sized chunks. Each task may need one or more stacks. Initially, one page is allocated to each stack. The use of the stack occurs in the usual fashion, with the top of stack pointer being modified at each push or pop operation. As the stack overflows, additional pages are allocated and linked with existing pages. As the stacks underflows into a page below, the page on top can be unlinked and put back into the free-page-list. In virtual memory, the mapping of a contiguous virtual space onto discontiguous physical pages is implicit in that a hardware mechanism automatically translates every virtual address into a physical one. The ELPS links are explicit in that no address translation is required. If it is implemented on top of virtual memory, then the links are virtual addresses. If there is no virtual memory, then the links are physical addresses.

4.1.2 Link Management

Figure 4.1 compares fix-sized stacks with ELPS variable-sized stacks. The free pages are linked together in the *free-page-list*. A processor requesting for a free page must lock the head of the list. To reduce contention for the lock, one free-page-list may be kept for each processor and a processor may be allowed to pick up a page from another processor's

free-page-list. Each page has two links, pointing to the page above and the page below it in the stack. Each page may also contain additional information regarding the page, such as page size or relative ordering of pages in a stack. The links and information fields may be stored separately in the page itself or combined together in a table. *Figure 4.2* shows two possible implementations. In both implementations, an address consists of two parts: the page number p and an offset to the data area d. In the first implementation (figure 4.2(a)), the links are stored together with the data in each page. In the second implementation (figure 4.2(b)), the links are collected in a central table, separate from data storage. An indexing scheme is needed to access the link table. We choose to implement the first scheme for the following reasons:

- Since the links for each page are accessed only by the owner task, distributed link storage allow interference free access to the links by parallel tasks, while centralized storage introduce unnecessary contention on the link table.

- Having the links together with the data would provide better cache performance. A task that accesses data at the bottom of a page is more likely to be the owner of that page, and thus would also need to access the links for that page. On the other hand, the centralized scheme would require the links of each page to be stored in a separate cache blocks to avoid extraneous contention on the cache block, thus wasting much of the space in each cache block.

4.2 Possible Implementations

There are three facets to the implementation of ELPS: (1) overflow and underflow detection, (2) overflow and underflow handling, and (3) data access. This section discusses the tradeoffs among the different possible implementations.

4.2.1 Overflow and Underflow Detection

Detection may be done purely in software or with hardware support. Software detection requires checking the value of the top of stack pointer after it is updated, but before any data is written into the stack. Should overflow (or underflow) occur, a simple subroutine call can be used to activate the overflow (or underflow) handler. The time cost of software detection depends on the frequency of the updates to the top of stack pointer.

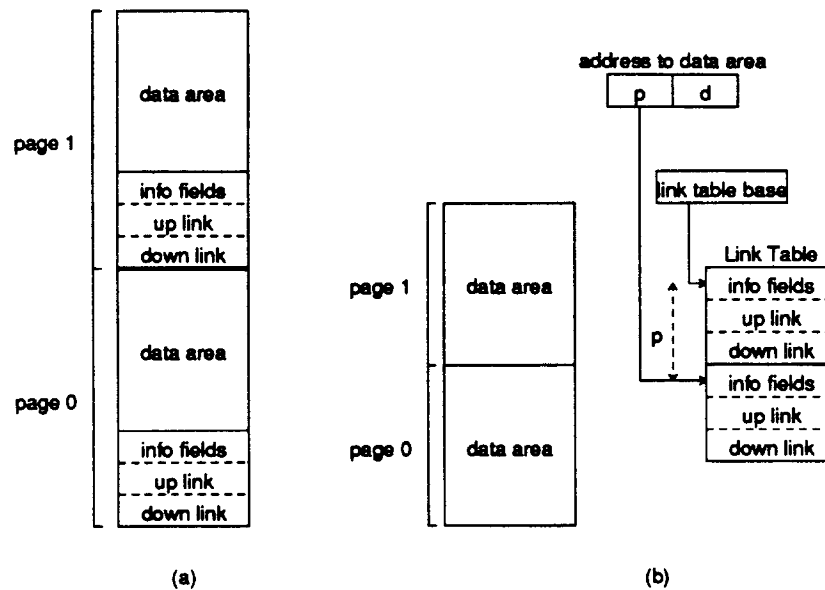


Figure 4.2: Two Link Storage Schemes for ELPS

Hardware support can be used to monitor access to a stack area or changes to the stack pointer. A special memory area may be designed to cause a trap to the overflow handler when it is written into. The PSI-II architecture [NN87] supports such a scheme, designating a part of memory at the top of the stack as the *gray page*. If all writes are restricted to the top of the stack, a write to outside the current page may be detected for underflow (locations above the top of the stack are not valid). In the WAM model, writes may occur to the middle of stack (e.g., variable bindings), and thus the gray page scheme can not handle underflow.

An alternative is to constantly monitor the value of the top of stack pointer, and generate a trap when it falls outside of the page. However, this can only work when the stack pointer is changed before new data is put into it. Depending on when the overflow signal becomes active, the system must be able to *undo* the stack operation that cause the overflow, and *repeat* it after a new page has been added to the stack and the top of stack pointer adjusted to point to the new page.

4.2.2 Overflow and Underflow Handling

Overflow handling involves requesting a page from the free-page-list and linking it to the existing pages. The top of stack pointer is updated to point to the bottom of the data area of the new page. Allocation of new pages may be done on demand (when the overflow occurs) or before the overflow can occur. On-demand allocation is simpler to implement and is as good or better than pre-allocation, considering the relatively low cost of overflow handling in ELPS and dynamic nature of stack space usage.

Underflow handling involves unlinking a page at the top of the stack and putting it back into the free-page-list. Deallocation may be done immediately when underflow occurs, or delayed until the free-page-list is empty (lazy deallocation). At this time, normal execution are suspended and processing power is used to scan the stacks for free pages not yet released. Lazy deallocation has the advantage that should the stack overflow again, request for a new page is not needed.

4.2.3 Data Access

By not allowing contiguous data objects to cross a page boundary, accessing data in an ELPS stack is identical to that of a contiguous stack. Contiguous objects such as activation records and data structures can be accessed using a base address and the field offset. The page size should be chosen to be much larger than the largest data structure. A very large data structure must be broken into smaller nested structures, with pointers from the main structure into the substructures. For linked objects such as lists, different elements may be stored in different pages, connected together by the links.

4.2.4 Address Comparison

Address comparison is often done to decide the relative positions of two data objects on the stack. If the pages in a stack are of arbitrary address ranges, it is not meaningful to simply compare two addresses in different pages.

The simplest approach is to restrict the page allocation algorithm such that a new page is selected only if it has a higher page address than the previous page of the stack. Then address comparison can proceed as if the stack address space were contiguous, since addresses across pages are monotonically increasing. This scheme increases the page

allocation and deallocation time, since a page with a higher address must be searched for in the free-page-list.

An alternative approach is to eliminate address comparison to allow the pages to be in arbitrary order. This will be discussed in section 4.3.3.

4.3 Qualitative Evaluation

4.3.1 Advantages

The dynamic memory management style of ELPS has a number of advantages over other schemes. They are:

- more efficient sharing of the global address space than static partitioning, reducing the need for garbage collection.
- much less expensive overflow handling than copy-when-overflow.
- more efficient handling of underflow.
- much simpler hardware support than virtual memory and does not require address translation which adds complexity to the cache system.

The heap style management may also be quite appropriate for garbage collection [TH88]. Garbage collection typically involves copying valid data to a new section of memory, and deallocating the old section which include invalid data. Due to the link structure in ELPS, any free page can be readily obtained for copying data from pages in current use, and pages in current use can easily be replaced by other pages.

4.3.2 Challenges

This heap-style management scheme introduces complexity which can potentially affect performance. This section considers the challenges of the scheme, and discusses the potential impact on performance. They are as follows:

- *Efficiency of overflow/underflow checking.* Without any hardware support, the efficiency of overflow/underflow checking depends largely on program behavior: the frequency in which items are put on and removed from the stack. The type of hardware support to be used depends on its effectiveness and implementation feasibility.

- *Frequency of page crossing.* Crossing a page boundary on an overflow takes additional time to follow the link. It is expected that the page size is chosen to be large enough such that overflow is infrequent. A potential problem would exist in the case where stack storage and removal occur frequently at around a page boundary.

- *Fragmentation.* In ELPS, there are two types of internal fragmentation that can occur. (Since the pages are of equal size and may be assigned to any task, there is no external fragmentation.) The first type of internal fragmentation occurs at the end of each non-top-most page, where some space is left unused because a contiguous data object does not fit and must be placed on the next page. The second type occurs at the end of the top-most page on the stack, the space is left unused when a task goes to sleep. If the page size is properly chosen, the internal fragmentation can be minimized and may be insignificant. Compared with static partitioning, internal fragmentation of ELPS pages is much smaller than internal fragmentation of the fixed address range. From that view, reduced fragmentation is the biggest advantage of ELPS.

- *Elimination of address comparison.* If the monotonic stack is too restrictive, the pages may be allowed to be in arbitrary order. However, this requires that address comparison be eliminated since comparing the address of data objects in different pages of the same stack has no significance. The following section will describe how it can be done for execution of Prolog.

4.3.3 Elimination of Address Comparison

If there is no address ordering among the pages, simple address comparison can not be done to determine the relative positions of objects on the same stack but in different pages. Therefore, address comparison should be eliminated. The following schemes may be used to eliminate address comparison in the WAM:

1. **Separate Environment and Choicepoint Stacks**
   If the control stack is used to store both environments and choicepoints, E points to the topmost environment, whereas B points to the topmost choicepoint. When new space is allocated on top of the stack, the two pointers are compared to find the top of the stack. By separating the environment stack and the choicepoint stack, each stack has its own top of stack pointer, and thus the comparison is no longer

needed. Separating the stacks also allows some environment space to be recovered, and enhances efficiency of garbage collection [TH88]. Under static partitioning, having an extra stack increases the chance of overflow because the shared space has to be further divided. This is not a problem under ELPS, because ELPS can support millions more stacks.

2. **Trail All Bindings**

   When binding an unbound variable, the variable address is compared with stack markers. If the variable is below B, the pointer to the last choicepoint on the stack, or below HB, the pointer to the heap, then the binding must be trailed. We propose two alternatives to the usual address comparison:

   (a) trail all bindings, thus eliminating the need for comparison;

   (b) if the addresses are on the same page, compare and trail only if variable lies below the pointer; otherwise, trail any way.

   Trailing all bindings can potentially double the amount of trailing [TD87], but significant saving is obtained by avoiding the address comparison. The time saving is significant when the address space is contiguous, and is even more significant when the address space is discontiguous. Alternative (b) provides a partial check for trailing, and reduces the number of trailings. Since trailing takes up only about 5% of total execution time, trailing all bindings for the sake of simplicity may be a good tradeoff.

3. **Put All Variables on Heap**

   A variable that is allocated on the stack is potentially unsafe since the environment trimming and tail recursion optimization may deallocate the variable location, resulting in a dangling reference. A put_unsafe_value instruction is used to copy this variable onto the heap, making the variable safe. In the current scheme, a variable may reside on either the heap or the stack, and thus a pointer comparison is made to find out where a variable is located. We propose that all variables be put on the heap, eliminating the need for this comparison. Dereferencing a permanent variable requires an extra level of indirection, but time and complexity are reduced since no comparison is need to determine where the variable resides.

Standard order builtins, such as @<, use address comparison to determine the order of creation of two unbound variables. In such cases, either the monotonicity of the pages must

be maintained, or a tag must be kept for the relative order of the pages in a stack.

4.4 Chapter Summary

In this chapter, we have presented ELPS, a hybrid heap-stack storage model for parallel execution on shared memory multiprocessors. At the global address level, the space is organized into a heap, which is a linked list of pages. At the local task space level, the pages are used to form logical stacks. The issues of page link storage, overflow/overflow detection, overflow/underflow handling, and data access have been discussed with some proposed solutions. A qualitative evaluation points out the promising features of ELPS as well as the challenges that needs to be overcome for efficient implementation.

In the following chapter, we will describe the multiprocessor simulation system used to evaluate ELPS. Details of the implementation of ELPS on the simulated multiprocessor will also be provided.

Chapter 5

NuSim: A Multiprocessor System Simulator

In the previous chapter, we described a hybrid heap-stack scheme for dynamic allocation of memory. In this chapter, we present the simulation methodology used to evaluate the proposed memory management scheme.

5.1 Introduction

Due to the numerous and intricate details involved in the operation of a computer system, simulation is an essential and effective approach in understanding and verifying theoretical models and architecture designs before they are built. There are many simulation methodologies, which differ in the level of details being simulated.

We have built a simulation system (called *NuSim*) to facilitate our studies of memory management for parallel execution on shared memory multiprocessors. This simulator framework allows for the complete system simulation: from the instruction set level to the memory architecture level with caches and communication protocols. The key feature of this simulator framework is flexibility, which allows for extensive instrumentation and continual updates and changes. The modular design identifies main features of the execution model and the architectures being simulated as cleanly separated modules with clearly defined interfaces. This allows for easy modifications to the individual modules to support new execution models and architectures.

Currently, the simulator supports the PPP Execution Model [Fag87], which ex-

ploits AND/OR parallelism in Prolog programs, and a *Multi* memory architecture [Bel85], multiple coherent caches on a single bus. The long term goal is to provide a flexible simulation environment where extensions to the execution model and the architecture can easily be incorporated, by modifying existing modules or replacing them with new ones. For example, the Aquarius II architecture ([DS88, Appendix 3] and [NS88]) and the multiple bus multiprocessor architecture [Car89] may be simulated using NuSim, with the appropriate replacement modules. The complete listing of NuSim code is available as a technical report [Ngu90].

5.2 Simulator Design Goals

The design goals for this simulator are:

1. **modularity**
   The simulator framework designed to be as modular as possible, with clean separation of the modules dealing with different features of the execution model and the architecture. This also allows easy modifications to the architecture and/or the execution model.

2. **simulation time efficiency**
   Simulation is inherently time consuming, thus it should take as little time as possible to simulate the model and the architecture running a benchmark. Simulation of a multiprocessor system on a sequential host can be several orders of magnitude worse than actual run time of the multiprocessor target.

3. **simulation space efficiency**
   The simulator should take up as little space as possible at run time, both the code space and the data space. Byte code is used for more realistic architecture simulation (taking into account code as well as data accesses) and also for greater space efficiency.

4. **good programming practice**
   Strict programming discipline is used in coding. Separate modules are isolated in different files, with local and exported functions explicitly declared (the default all global style of C is not used). This is important in catching bugs at an early stage, making the code easily understandable to others and safely modifiable (by data hid-

ing). Macros are used where appropriate to increase code legibility and high level understanding, to reduce repetition, and to make fast and reliable changes possible. For example:

```
#define X_FIELD 5
#define PR_STAT(a,b) fprintf(fp, "A = %5d ; B = %7.1f%%", a, b)
```

X_FIELD is more logical than 5. PR_STAT may be reused many times, reducing the amount of code text, and the print format change can easily be made at only one place.

5. **portability**
The simulator is written in fairly portable C code, with the exception of the assembly language routine for coroutining, and a few system dependent operating system calls (needed for resource usage monitoring of the host machine). It was developed on a Sun 3/50 (MC68020) running 4.3 BSD Unix^{TM}. It has been ported to the VAX/785 and VAX/8600 (running 4.3 BSD Unix^{TM}) and an Intel 386 personal computer (running System V Unix^{TM}). It should also be portable to other 32-bit machines. Porting to non-Unix machines require some changes to the system calls.

5.3 Simulation System Overview

5.3.1 Program Transformation

A Prolog benchmark goes through several stages of transformation before it gets to run on the simulator. *Figure 5.1* shows these steps. First, the Prolog benchmark gets annotated by the programmer for parallel execution (static scheduling approach). This programmer's annotation can be assisted by use of the Static Data Dependency Analysis program (SDDA) [Cha85, Cit88]. Some work has been done by Bitar [Bit89a] to automatically annotate Prolog programs, using data dependency information obtained from SDDA. Second, this annotated Prolog program gets compiled by the PLM compiler [VR84]. The PLM compiler is capable of accepting annotations for AND-parallelism (e.g., `a :- b & c`) and OR-parallelism (e.g., `orpar(a)`). The compiler transforms the annotated Prolog code into PPP assembly language code. The simulator directly loads in assembly code, then

activates the assembler to assemble it into a *byte code* stream and to generate a symbol table. The assembler is integrated with the main simulation engine to eliminate the need to write out the symbol table to a file and to re-load it in for execution.

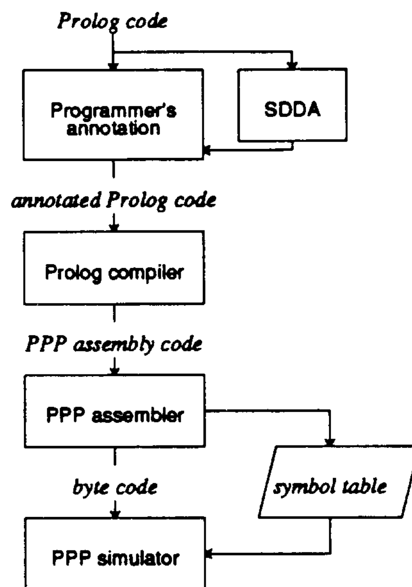


Figure 5.1: Program Transformation

The following is an example of the program transformation.

Prolog program:

```
main :- a(X), b(Y).

a(X) :- ...
a(X) :- ...

b(X) :- ...
```

Annotated program:

```
:- option(par, orpar(a/1)).   % OR parallelism annotation

main :- a(X) & b(Y).          % AND parallelism annotation

a(X) :- ...
a(X) :- ...
```

```
b(X) :- ...
```

Compiled PLM assembly code:

```
procedure  main/0
        i_allocate  2,_1_206,_1_206,0
        put_variable  X1,X1
        call_p  a/1,1,1          % AND-fork procedure 'a'
        put_variable  X1,X1
        call_p  b/1,2,1          % AND-fork procedure 'b'
        wait  1                  % join
        deallocate
        proceed
_1_206: 2                        % Join table
        0

procedure  a/1
        try_me_else_p  _2_776           % OR-fork 1st clause 'a'
                ...
        proceed
_2_776: trust_me_else_p  fail           % OR-fork 2nd clause 'a'
                ...
        proceed

procedure  b/1
                ...
        proceed
```

5.3.2 Design Considerations

In designing a new simulation system, we had considered a wide variety of options. Our chosen simulator design and implementation methodology is a compromise of the various options to be most suitable to the goals described in section 5.2. We prefer detailed simulation of the complete system architecture over trace simulation and stochastic modeling to obtain more accurate measurements on system memory performance. We chose an *event driven simulation* approach over *a cycle by cycle simulation* mainly because it has been proven to be quite successful with previous multiprocessor simulation efforts. SIMON [Fuj83a, Fuj83b] is a simulator for multicomputer networks and Multisim [CHN88] is a simulator for single bus cache coherency protocol. We also believe that the event driven approach allows for a more modular and hierarchical design.

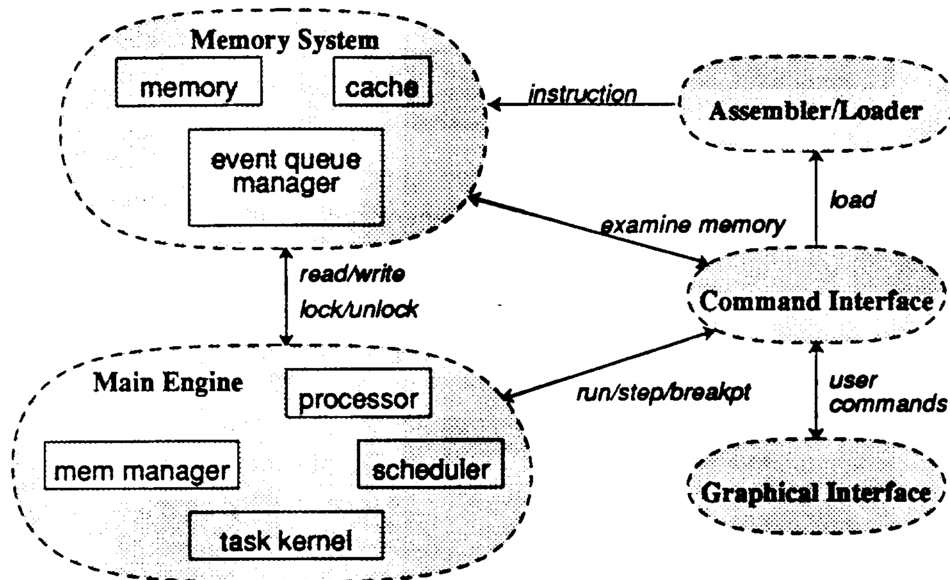


Figure 5.2: NuSim Simulator Framework

We chose C as the implementation language over the specialized languages such as Endot ISP [End87] and GPSS [Scr74] for practical reasons: C is fairly well understood, flexible, portable, and efficient. It also allows for easy integration with existing modules of simulators such as Multisim and SIMON, which were both coded in C.

5.4 Module Description

The simulator is basically an event driven simulation system, with memory accesses as the events. The events are ordered by time stamps and access priority, according to the network arbitration protocol.

Figure 5.2 shows an overview of the simulator, which consists of: the assembler/loader, the command interface, the graphical interface, the main simulation engine, and the memory system.

5.4.1 Assembler/Loader

The *assembler/loader* loads in the assembly language file, assembles it into a byte-code stream, and stores the stream into the code space of the simulated memory. This

module also builds a symbol table to be used during execution of the code.

5.4.2 Command Interface

The *command interface* allows for user interaction with the simulator. The user commands can be entered *interactively* at a terminal, or *collectively* provided in a command file which is read in and executed. This command interface is particularly useful for interactive debugging, which will be discussed in section 5.6 (Multi-level Debugging Facility).

5.4.3 Graphical Interface

The *graphical interface* provides a more user-friendly interactive environment for debugging and observing the execution activities of the simulated multiprocessor system. It may also be used to graphically display performance statistics. Section 5.7 will provide an extended description of *xNuSim*, a graphical interface for multiprocessor simulators.

5.4.4 Main Simulation Engine

The *main simulation engine* is composed of submodules which simulate the parallel execution model on the processors. The submodules are: the *processor*, the *task kernel*, the *scheduler*, and the *memory manager*.

Processor

The *processor* submodule contains routines to emulate the model processor, executing a specified instruction set. Currently, the VLSI-PLM [STN*88] instruction set is supported by the processor module. All the PPP Execution Model instructions for parallel execution (e.g., `call_p` and `try_p`) are also supported. The width of the registers is 32-bit, which includes 2 bits for tags.

Memory Manager

An efficient memory management scheme is needed to support the PPP execution model, allowing efficient allocation and deallocation of memory space for the PPP's potentially numerous tasks. The *memory manager* submodule implements the Explicitly Linked Paging Stack (ELPS) memory management scheme described in the previous chapter. This

includes routines to service the page crossing upon stack overflow/underflow, and to manage the *free-page-list*. In the processor submodule, changes (increment/decrement) to a top of stack pointer is constantly monitored for potential page crossing. This simulates the boundary checking done by various software and hardware techniques. These operations will be described in greater detail in section 5.9.

Task Kernel Module

The *task kernel* submodule represents the parallel execution model. Currently, the PPP Execution Model [Fag87] is simulated. A task is a piece of work which may be executed in parallel with other tasks. In the PPP execution model, there are two types of tasks: AND-task and OR-task. In Fagin's thesis, these tasks are referred to as *processes*. To differentiate a PPP light-weight (shared address space) process from a heavy-weight Unix process, this entity is now called a task.

The task kernel submodule consists of routines to handle task creation, communication and termination. The PPP model supports independent AND parallelism, OR parallelism, and semi-intelligent backtracking. The simulator currently does not support semi-intelligent backtracking. Studies by Fagin indicate that few Prolog programs can take advantage of semi-intelligent backtracking. Therefore, it is left out mainly to reduce the complexity of the simulator and to focus on memory management for AND- and OR-parallelism. Sequential execution stands to gain the most from semi-intelligent backtracking.

The PPP's task communication is implemented using shared memory, with each task having a communication area. *Figure 5.3* shows *task A* communicating with *task B*. If the receiving task is executing in some processor, the sending processor writes into a special memory location associated with the receiving processor, causing an interrupt on the receiving processor (*figure 5.3(a)*). If task B is sleeping, the transaction will be put in the communication area of its task control block, which will be noticed when the task gets picked up by an idle processor (*figure 5.3(b)*).

The task kernel submodule also includes routines to manage the multiple binding environments for OR-tasks. The current implementation supports the dynamic window linking scheme for AND/OR parallel execution of the PPP Model. These routines include: *bind* (to store data in these hashwindows), *dereference* (to retrieve data), and handler routines for success and failure (to link and unlink the window chain).

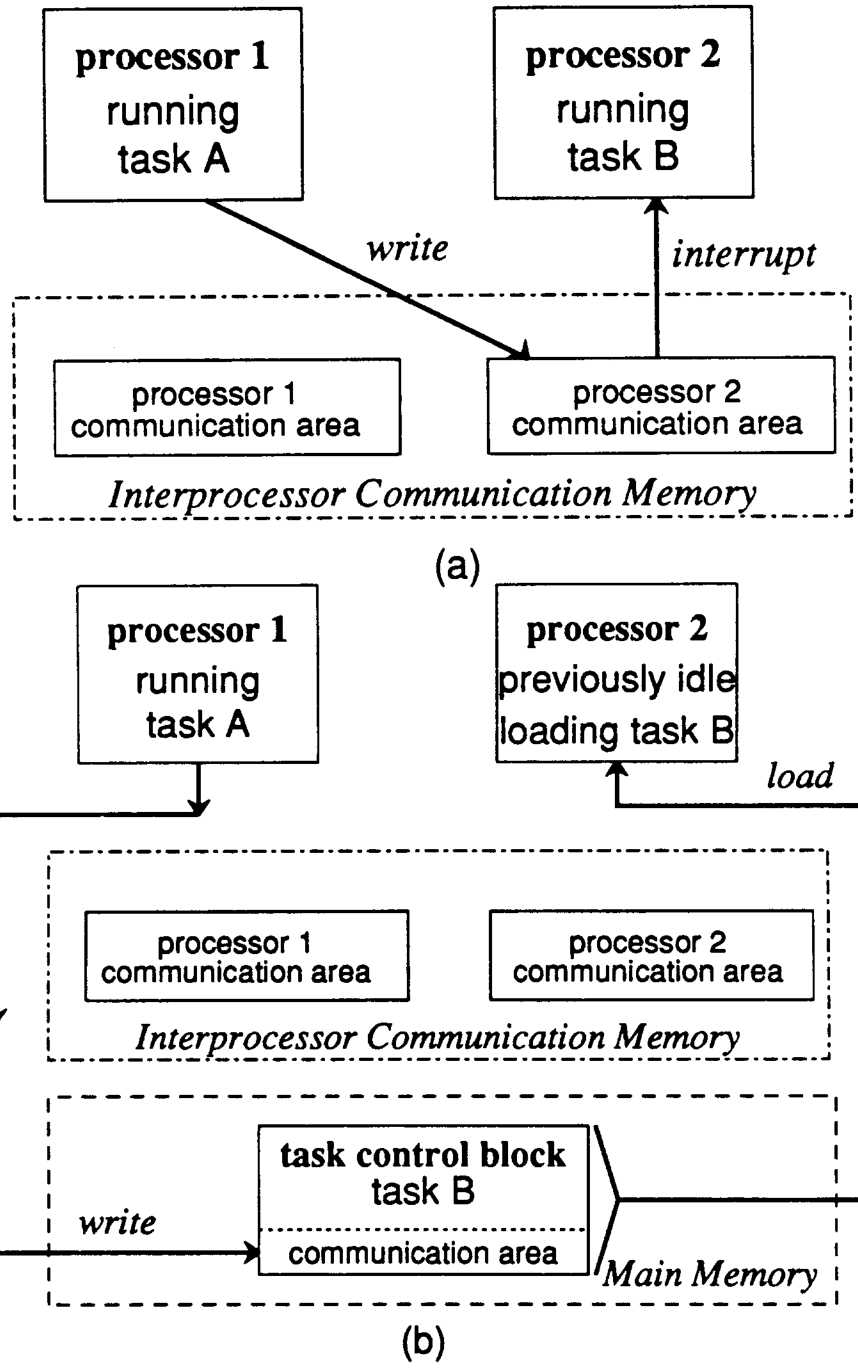


Figure 5.3: Task Communication

Scheduler

The *scheduler* manages the task pool and assign tasks to processors. For the PPP Execution Model, all tasks get spawned as compiled, unless there is insufficient resource. Currently, a naive scheduler allows an idle processor to randomly pick the next ready task to run.

The scheduler operates on a set of queues. The current implementation has four queues. The *ready queue* contains the tasks that are ready to be run. The *sleeping queue* contains tasks that have succeeded and returned answer to its parent. A task must be retained in sleeping state for the storage of its computed result, and for the state which may be backtracked into at future time. The sleeping queue also contains parent tasks that are waiting for responses from the children. The *pending queue* contains tasks that are suspended due to an I/O request. To maintain standard Prolog semantics, these tasks are not allowed to proceed until they become the leftmost child. Finally, the *free queue* contains task table entries that are unused, available for the creation of new tasks.

5.4.5 Memory System

The *memory system* simulates the memory subsystem of the architecture which is composed of: caches, interconnection network, and main memory. As shown in *figure 5.2*, the interface to the main engine is through four memory access routines: *read*, *write*, *lock*, and *unlock*. Lock and unlock provides the synchronization primitive needed to achieve mutual exclusion.

Multisim [CHN88] is a simulator for a single bus multiprocessor with multiple hardware coherent caches (*figure 5.4*). Three modules have been extracted from Multisim and integrated into NuSim to provide a memory system for the new simulator: the event manager, the cache module, and the memory module.

The *event manager* contains routines to manage (insert and delete) the prioritized event queue, simulating the single bus broadcast and arbitration protocol for the requests coming from the caches. It also contains the system-dependent assembly language routines `save_state()` and `restore_state()`, which are called to switch among the processors being simulated.

The *cache module* simulates the cache of a processor, responding to memory requests coming from the processors. In case of a cache miss, the cache module sends out the

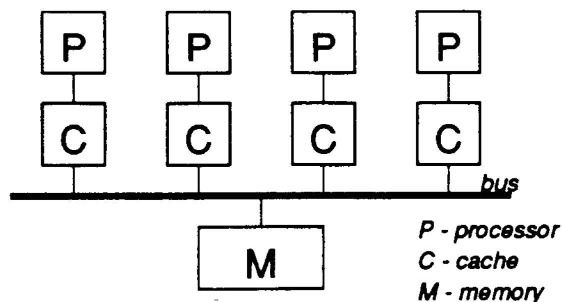


Figure 5.4: The Multi Architecture

memory request onto the bus, and awaits response. The operations include cache lookups, bus request, bus broadcast, cache busy wait, and other steps needed to implement the coherency protocol. Multisim implements Bitar's cache coherency protocol with the *cache lock state* [BD86].

The simulated processor has a 32-bit internal datapath. Since 3 bits are used for tags[1], there are 29 bits for storing data. The *memory module* of Multisim simulates the multiprocessor's 29-bit word-addressable main memory. It uses a 2-level paging scheme similar to that of virtual memory managed by the operating system. This paging scheme allows the simulated architecture to access the entire 29-bit address range, while using a much smaller percentage. Space is allocated only for simulated memory blocks which are actually read or written by the simulated processor. This is an important feature since it automatically manages the address space, and frees the programmer from the concern of mapping the benchmark code and data spaces onto the limited available space allowed by the operating system (4.3 BSD or System V Unix^{TM}) for each Unix process. C functions `calloc()` and `malloc()` are used to obtain memory dynamically from the operating system.

5.5 Instrumentation

The major advantage of a simulation over an actual implementation is the ease and flexibility of instrumentation. Software instrumentation in a simulator can be done much easier than hardware instrumentation in an actual implementation. The main goal of

[1] 2 bits for the 4 main Prolog types (variable, constant, list, and structure) and 1 bit for the symbol table entry subtype.

our simulation system is to be able to study the behavior of the parallel execution model as well as the performance of the underlying architecture supporting the model.

With respect to the PPP parallel execution model, the simulator is instrumented to record:

1. parallel execution overhead: task creation, termination and communication.
2. memory space usage and performance of memory management scheme (ELPS): maximum code and data space required and cost of allocation, deallocation, and redistribution.
3. degree of sharing: frequency of accesses to shared data, and size of shared data.

With respect to the system architecture, the simulator is instrumented to measure:

1. processor performance: percentage of memory accesses versus internal operations.
2. cache hit ratio, as a function of cache size, block size, and associativity.
3. the effectiveness of the cache coherency protocol, as a function of the cache hit ratios, number of cache updates, invalidations, and bus utilization.
4. the effectiveness of the synchronization primitives and the locking protocols, as a function of the time for busy waits of locked objects.

Instrumentation is done by putting counters at appropriate places in the simulator routines. The results are reported at the end of execution of a benchmark. For some measurement such as lifetime of a task, the result will be reported as it becomes available. The desired performance numbers, such as percentage of reads versus writes, are computed from these counter values. The *performance table generator* accepts the counter values as input, computes the desired performance numbers, and prints them in a table format. A spread sheet program is also used in analyzing data and printing out tables.

The simulator itself was profiled to determine the routines where most of the CPU time are spent. This information helps in optimizing the critical routines in speeding up the simulation time. Currently, there are two different sets of assembly language routines (`save_state()` and `restore_state()`) to perform the coroutine switch. One set copies a portion of the C execution stack to a *saved area*. This technique works with *DBX*, the Unix symbolic debugger, to allow debugging of the simulator. The second routine uses *multiple*

stacks and swap the stack pointers for a coroutine switch. This gives 30% faster execution but does not work with DBX. Fortunately, it does work with *GDB*, the GNU symbolic debugger from the Free Software Foundation. The decision as to which set of routines to use is made at compile time.

5.6 Multi-level Debugging Facility

NuSim contains an interactive debugger to help debug the simulated system as well as the simulator itself. This debugger allows for setting various types of breakpoints at the *instruction level* and for printing out special Prolog data structures such as choicepoints, environments, and task table entry. With the appropriate settings, the NuSim debugger can interact with a C language symbolic debugger (such as *DBX* or *GDB*) to allow multilevel debugging of the execution model and the C source code (which represents the microcode of the hardware engine). Please see NuSim User's Manual (*appendix A*) for a more detailed explanation of the interaction.

The following three figures provide a "feel" for the multi-level debugging capability of NuSim. First, figure 5.5 shows a sample run of the simulator with the NuSim debugger. After the compiled Prolog program has been loaded, `ml` shows the memory layout of the target multiprocessor's memory, and `code` shows a listing of the program, and `bp` sets a breakpoint at a specified code address. Simulation is then started by `run`, and the debugger prompt appears when the breakpoint is reached. `ps` shows the processor state (register values) of the specified processor.

Figure 5.6 shows how debugging can be accomplished at the C language symbolic level. GDB is the symbolic debugger used in the illustration. And thirdly, figure 5.7 shows multilevel debugging with both the NuSim debugger and GDB.

In addition to the basic status checks and breakpoint capability, other debugging commands are added as necessary. A menu system with table driven input is set up to allow easy addition of new debugging commands. The main goal in debugging support is to minimize the user input, letting the debugger do most of the work, and to keep the outputs at a logical, abstract level. Hiding much of the detailed information makes it easier to understand the status of the execution. For example, `print topmost choicepoint` is used instead of `print choicepoint starting at <addr>` since the first is more logical, and does more work (it must first check for the top of choicepoint stack pointer).

```
UNIX> nusim Benchmarks/Misc/con1.w
***** NuSim ----- SUN Version 1.2 ----- July 8, 1989 (fast coroutine)
    File '/hprg/NuSim/Benchmarks/Misc/con1.w' loaded [0x3820-0x3872].
Type 'h' for help.
--NuSim:DBG> ml
*** MEMORY LAYOUT  [0x1000 - 0x1fffffff] ***
  FTQ base/head/tail  0x1020/0x3f/0x0
                ...
     Task Table base  0x1120
       S/A-Heap base  0x1b20
      Code start/end  0x3820/0x3872
           Code size  83 words
     DataSpace start  0x3880
      Task Data size  8388381 words
          Heap ratio  0.40
                ...
         Window size  128 words
--NuSim:TOP> code 0x3820 0x3825
    0x    3820: put_list              X0
    0x    3822: unify_constant        a
    0x    3824: unify_variable_x      X3
--NuSim:TOP> bp 0x3824 b
Breakpoint 0x3824 set!
--NuSim:TOP> run
  (P0 T0) *  0x3824  unify_variable_x      X3
--NuSim:DBG> ps 0
proc #0 executing task #0 at time 380 -- timer = 0, cflow = FORWARD
    P:     3826      CP:         0      E:         0      B:   336ba5
   TR:   80379c       H:      3901     HB:      3900      S:     3900
   TS:   59d1a0      oP:      3824    cut:         0   mode:    write
   A0:     3900      A1:         0     A2:         0     A3:        0
   A4:        0      A5:         0     A6:         0     A7:        0
   t0:        0      t1:         0     t2:         0     t3:        0
   t4:        0      t5:         0     t6:         0     t7:        0
--NuSim:DBG>
```

Figure 5.5: Simulation Run with NuSim Debugger

```
UNIX> gdb nusim
GDB 2.7, Copyright (C) 1988 Free Software Foundation, Inc.
Reading symbol data from nusim...done.
(gdb) break init_sim
Breakpoint 1 at 0x13dea: file init_sim.c, line 46.
(gdb) run -d Benchmarks/Misc/con1.w
Starting program: nusim -d Benchmarks/Misc/con1.w
***** NuSim ----- SUN Version 1.2 ----- July 8, 1989 (fast coroutine)
Bpt 1, init_sim (pid=0) (init_sim.c line 46)
46          if (!(reqs[pid] = (MEM_EVENT *) malloc(sizeof(MEM_EVENT))))
(gdb) info break
Breakpoints:
Num Enb   Address    Where
#1    y  0x00013dea  in init_sim (init_sim.c line 46)
(gdb) delete 1
(gdb) next
49          if (!(procs[pid] = (PROC_STATE *) malloc(sizeof(PROC_STATE))))
(gdb) cont
Continuing.
    File '/hprg/NuSim/Benchmarks/Misc/con1.w' loaded [0x3820-0x3872].
[a,b,c,d,e]
Top level query success
Exiting Simulator... Simulated time: 1339 cycles
RUSAGE:  2.0u  0.7s  1388+4329+0k  6+1io  10pf+0w
(gdb)
```

Figure 5.6: Simulation Run with GDB (C-language) Symbolic Debugger

```
UNIX> gdb nusim
GDB 2.7, Copyright (C) 1988 Free Software Foundation, Inc.
Reading symbol data from nusim...done.
Breakpoint 1 at 0x1b4fe: file toplevel.c, line 124.
***** NuSim ----- SUN Version 1.2 ----- July 8, 1989 (fast coroutine)
    File '/hprg/NuSim/Benchmarks/Misc/con1.w' loaded [0x3820-0x3872].
Type 'h' for help.
--NuSim:TOP> bp concat/3 b
Breakpoint concat/3:0x3852 set!
--NuSim:TOP> run
  (P0 T0) * concat/3:  0x3852  switch_on_term       _2_719, _2_720, fail
  (P0 T0) (884) Call: concat([a,b,c],[d,e],_59d1a0)
--NuSim:DBG> dbx
Bpt 1, dbx_break () (toplevel.c line 124)
124     }
(gdb) break switch_on_term
Breakpoint 5 at 0xd1f2: file /vlsi2/tam/NuSim/Source/Proc/index.c, line 21.
(gdb) c
Continuing.
--NuSim:DBG> c
Bpt 5, switch_on_term (pid=0) (/vlsi2/tam/NuSim/Source/Proc/index.c line 23)
23          DECLARE_proc;
(gdb) next
25          T(0) = dereference(pid, A(0));
(gdb) n
26          switch( TAG(T(0)) ) {
(gdb) ndb
--NuSim:DBG> env 0
  Sequential environment (base = 59d1a5)
     E:        0      CP:         0       B:   336ba5
    Y0: 8059d1a0      Y1:         0      Y2:         0      Y3:         0
--NuSim:DBG> c
(gdb)
```

Figure 5.7: Multi-level Debugging with NuSim Debugger and GDB

5.7 xNuSim: A Graphical Interface for Multiprocessor Simulators

A graphical interface can greatly enhance the ease of use of a simulator, and make it easier to monitor the various activities of the simulated architecture. A graphical interface can also be used to report performance results. The process of displaying intermediate simulation data is also known as *animation* in simulation terminology [Sar88].

A graphical interface for multiprocessor simulators has been developed and integrated with NuSim. It runs under the *X11 Window System* and is thus called **xNuSim** [Pan89]). This graphical debugging environment is modeled after DUES [Wei88], the graphical interface to the sequential VLSI-PLM simulator. xNuSim provides multiple windows for viewing of code, execution output, processors' status, and memory contents. It enhances the ease of use of the NuSim simulator.

The key feature of xNuSim is its *loosely coupled interface* with the NuSim simulator, thus enabling it to be used with other simulators as well. xNuSim knows nothing about the internal operations of NuSim and only communicate with NuSim via the NuSim's *command interface* (please see figure 5.2 and section 5.4.2). When the simulator is run without xNuSim, commands are entered from keyboard and simulator outputs are shown on the screen in text form. With xNuSim, commands may be entered either by use of the mouse and pop-up menus, or by use of keyboard as before. xNuSim interprets the output of the simulator to extract the relevant data used in the graphical display windows. To use xNuSim with another simulator, only simple changes to the menu tables and command formats are needed to customize xNuSim for that simulator. *Figure 5.8* shows a sample setup of the graphical interface.

For future work, additional features can be added to this graphical environment to allow monitoring of parallel tasks and multiprocessor activities. A useful feature would be to show the execution tree as time progresses. For example, the graphical environment was implemented at the Argonne National Lab for their OR-parallel system [DL87]. This graphical environment accepts execution traces as inputs and shows the changing of the execution tree through time. Such a system for the PPP model would be quite useful in understanding parallel execution and the effects of the scheduler.

Figure 5.8: A Sample Setup of the xNuSim Graphical Interface

5.8 Compatibility and Extendability

The simulator framework is designed to be compatible with many of the existing software packages, improving existing ideas and allowing for better integration. The simulator framework is also designed to accommodate future changes with the least possible amount of programming effort. The following are some notes on compatibility and extendability of the simulator framework:

1. The PLM Compiler with the latest support for PPP is used to generate PPP code from a Prolog benchmark.

2. Due to removal of CDR-coding and split environment and choicepoint stack, the semantics of a few instructions have been slightly altered. Thus some routines in the library of builtins in VLSI-PLM instruction set would require rewriting, especially those which contain a sequence of instructions to build up a list or structure. These builtins are currently written in C code, as is the PLM instruction set, thus simulating the builtins in microcode. These builtins may be written in VLSI-PLM code in the future to simulate the builtins as software routines.

3. The simulator is fully compatible with the Multisim memory and cache module. Multisim has been incorporated into the new simulator. With the interface of the four routines to read, write, lock, and unlock, a memory and cache module for another memory architecture may replace the single bus hardware cache coherent system. The simulator should also be compatible with other memory system simulators such as SIMON [Fuj83a] which satisfy two requirements: that they employ event driven methodology, and that they are coded in C (or object code compatible).

4. Ideally, it is desirable to have other processor modules to be compatible with the PPP Task Kernel. Compatibility works best at a high level abstraction. For example, `push(H,X)` is being used instead of `memwrite(H++,X)`. This allows the `push` function to increment or decrement the pointer appropriately, and the checking of page boundary crossing can also be done in the push function.

5. The ELPS memory management module should be compatible with any instruction set, since page crossing exceptions are handled at a low level below the instruction set.

6. Any modification to the binding environments involve the *basic* routines, such as bind, dereference, trail and also operations upon process creation, termination (success or failure), and switching (success and wakeup).

7. There is at least one simple extension to the naive scheduler of the PPP: attach a priority to each process and keep the ready tasks in priority queue or tree structure. Idle processors can select the ready task with the highest priority. More extensive changes to the model would require additional data structures for handling of the parallel goals, and appropriate routines to manipulate them. Changes in the scheduling scheme would affect task creation, termination, and switching.

5.9 Implementation of ELPS on the Simulated Multiprocessor

Two different techniques, one with hardware support and another entirely in software, are used to implement ELPS on the simulated multiprocessor architecture. The dataspace address range for all tasks is partitioned into small pages of a power of 2 size. A small block of the lowest addressed words in each page is reserved for storing the links and information regarding the page. Currently, the two lowest words are used to store the two links.

5.9.1 Software Checking

In the software technique, overflow checking is done on the stack pointer each time it is updated. The *page_mask* is a constant that used to obtain the page number. The overflow checking algorithm is as follows:

```
new_ptr      = stack_ptr + object_size;
current_page = stack_ptr & page_mask;
new_page     = new_ptr   & page_mask;
if (current_page != new_page) then
     call overflow_handler();
```

When overflow occurs, the overflow handler is called to obtain a new page, link it with the top page on the stack, and update the *stack_ptr*. This software check costs four cycles. It can

be reduced to two cycles if a register is designated to store the page boundary, containing the highest address in the current page. One *upper_page_limit* register is needed for each stack, and it is updated at the time of overflow. The checking is simplified to:

```
new_ptr        = stack_ptr + object_size;
if (new_ptr > upper_page_limit) then
     call overflow_handler();
```

To check for underflow, the stack pointer is compared against the boundary of the link fields, stored at the bottom of a page. A *link_mask* is used to determine the boundary of the link fields. The algorithm is similar to overflow. With a register reserved for the *lower_page_limit*, underflow checking can be done in two cycles.

5.9.2 Hardware Support for Checking

With hardware support, overflow and underflow checking can be overlapped with the stack push/pop operation, thus requiring no extra time. Two status signals are used to indicate an overflow (or underflow) and causes a trap to the exception handler. The *outside current page* signal indicates that the new stack pointer is outside of the previous page, which may be an overflow or an underflow, and the *inside link area* signal indicates that the new stack pointer is inside the link area, which is an underflow. *Figure 5.9* shows the hardware logic circuit, using a *page mask*, a *link mask*, and two comparators. The hardware support for this check requires the delay of one additional AND-gate and one OR-gate over a normal comparison. However, this should not lengthen the cycle time since equality comparison is generally not in the critical path of ALU designs. The figure uses the generalized case of the full word width. By fixing the page size or the link fields to a certain range, full word (e.g., 32-bit) comparison is not needed, thus reducing the chip area and comparison circuit delay in the processor. In the most general case, page mask and link mask can be writeable special registers, allowing for the configuration of the page size and the info associated with each page at the start of execution.

Page allocation is done on-demand at the time of overflow, and deallocation is done lazily. For fast startup time, only a portion of the dataspace is partitioned into pages and put into the free-page-list. The others are kept as uninitialized memory, and gets initialized only when the free-page-list is empty. If the uninitialized memory is also empty, normal

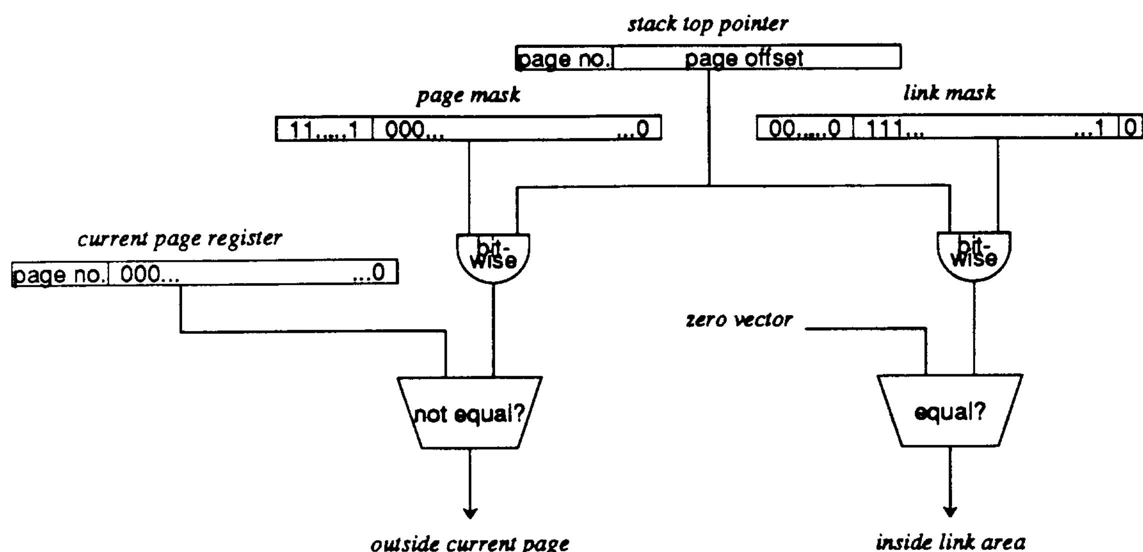


Figure 5.9: Hardware Support for Out-of-Page Check

outside current page ← pageno(*stack top pointer*) ≠ *current page register*

inside link area ← pageoffset(*stack top pointer*) < linkbound(*stack top pointer*)

execution in all processors are suspended, and the stacks are scanned for free pages not yet deallocated.

5.10 Chapter Summary

In this chapter, we describe a multiprocessor simulation system, called *NuSim*. It is an event-driven simulator with memory accesses as events. The modules are designed to simulate the processor, the memory system (including caches), and the interconnection network. The simulator accepts a compiled Prolog program as input and outputs performance evaluation statistics. The current implementation simulates a *multi* [Bel85], using the *cache lock state* protocol [BD86] to keep the caches consistent. Debugging support for the simulator includes a graphical interface, called *xNuSim* [Pan89], and a multi-level debugger (for the instruction level and the register transfer or microcode level). The ELPS memory management model is implemented on the simulated target multiprocessor using two schemes: software only and hardware support. The check for page boundary overflow costs two cycles for the software only scheme and require no extra time with hardware

support.

Chapter 6

Simulator Validation

6.1 Introduction

Simulation can be an accurate and effective approach in predicting performance of a new multiprocessor system, if the many intricate details in the hardware and software designs are taken into account. The degree of accuracy depends on how much detail is included in the simulator. To ensure that the simulator accurately reflects the system yet to be built, the simulator must be carefully validated for correct functional as well as timing results.

The validation process is carried out by comparing performance data from the new simulator with known data obtained from previously validated sources. The validation process itself can be quite tedious and difficult, with massive amounts of information that need to be analyzed. In this chapter, we present our approach to validating the framework and the processor module of NuSim, described in the previous chapter. The process involves sequential execution of benchmark programs on NuSim and a uniprocessor simulator, comparing results and performance data.

6.2 Validation Methodology

There are many approaches to the validation of a simulation model [Sar88]. The concept of our approach to validation is quite simple: comparing new, unverified results with previously known answers. The more difficult task is the careful consideration of the many different factors that can affect the results and the degree of these effects. The

validation process for a computer system simulator is best done in a stepwise fashion. The exact details of the necessary steps depends on the availability of the known result, or the *basis*, used for comparison.

For the rest of this thesis, the term *host* designates the machine on which the simulator is run and *target* refers to the computer architecture/system being simulated. Validation refers to the process of ensuring that the simulator is coded correctly and that it accurately models the target.

In the initial phase, where a paper design is the only basis available, validation of the simulator usually consists of:

1. *Manually* checking for correct coding according to the paper design.

2. Running the simulator and checking for functional correctness, comparing the results with *manually* worked out solutions.

3. *Manually* checking the timing of sub-blocks in the simulator.

4. Running the simulator to obtain timing estimates.

5. Running simulator with instrumentation turned on to capture dynamic execution statistics.

The term *manually* used above refers to the *ad hoc* approach of eyeballing (for steps 1 and 3), hand calculations (step 2), or writing small, very special purpose software tools to accomplish the tasks. This approach is tedious and error prone, but is often the only possible way at this phase since a paper design is the only available basis. In step 5, the monitor facility for instrumentation should not affect the timing.

Once the initial simulator is validated, it may be used as a basis for validating other simulation systems. The validation process can now be done with a greater degree of automation, and thus achieving greater efficiency. However, great care must still be taken to understand the factors that cause discrepancies.

The validation process of a multiprocessor system[1] simulator involves the following steps:

[1]The term *multiprocessor system* is used to include both the multiprocessor architecture and the parallel execution model

1. *sequential execution on one processor.* This is done to test the processor module of the simulator and the relevant support modules such as assembler and loader.

2. *parallel execution on one processor.* This is a degenerate case, done to measure the overhead of parallel execution.

3. *parallel execution on two processors.* This is a special case for testing interprocessor communication with no interference since there is exactly one sender and one receiver.

4. *parallel execution on three or more processors.* This is the general case of parallel execution, with potential for interference on shared resources such as the memory and communication channels. It is also used to test the full extent of the parallel execution model. As more processors are added to the configuration, the saturation of shared resources will occur and bottlenecks will appear.

In this chapter, we present the application of the first step of validation of a multiprocessor simulator, using a previously validated uniprocessor simulator as a basis (this first step is the foundation for the other three steps: the simulation result of each of the three steps is compared against the result of the first step). Since there are architecture and execution model variations in the two simulators, their results are compared for *proximity*, not for exact *equality*. The following sections provide details on the simulators and the validation approach.

6.3 Simulator Descriptions

To validate NuSim, we use a previously validated simulator, called *VPsim*, as a basis for comparison. Both simulators provide an abstract machine engine for fast execution of the Prolog language. This section provide a description of VPsim and its similarity and differences to NuSim.

6.3.1 VPsim

VPsim is a register transfer level simulator for the VLSI-PLM [Hol88, STN*88]. This chip is a VLSI implementation of a high performance engine for Prolog, a modified version of the abstract machine proposed by Warren [War83]. VPsim is written in the C

language, consisting of 4500 lines of C code and 9000 lines of microcode operations (register transfers, CPU operations and microbranches).

To verify VPsim's functional correctness, a wide variety of Prolog programs was run on VPsim and their results were compared with those obtained from runs on commercial Prolog systems such as Quintus Prolog. Because VPsim is microcode driven, the microstates automatically provide accurate timing, with each microstate being executed in exactly one processor cycle. Gate and transistor level simulations of the VLSI-PLM chip are compared against the results from VPsim. In the final stage, the chip is tested. It passes an extensive testing process and successfully executes a number of benchmark programs.

From the perspective of our research, VPsim is a solid simulator that has been well tested and has been verified by the hardware. It is an available resource that can be used as a basis for testing other simulation systems.

6.3.2 Simulator Differences

Although both NuSim and VPsim essentially simulate the VLSI-PLM chip, they were created for very different purposes. VPsim was designed as a simulator for a very specific microarchitecture of a Prolog processor. Details of the VLSI-PLM microarchitecture are "hard-wired" into the microcode, in terms of what micro-operations are possible and the constraints in packing the micro-operations into a micro-state. On the other hand, NuSim was conceived as a more general purpose multiprocessor simulator for *system integration*, dealing at all levels from hardware architecture to software execution model. It will be used to experiment with different architectures and execution model tradeoffs.

Because of the different goals in creating the simulators, there are a number of differences between them. These differences are identified to help us understand the differences in performance numbers. The following are some differences between VPsim and NuSim (running sequential code):

- **simulation level.** VPsim is a register-transfer-level, cycle-by-cycle simulation, while NuSim is an event driven simulator which steps by memory access. The clock of VPsim is incremented each cycle, while the clock of NuSim is incremented by a value obtained from table lookup plus the simulated memory access time.
- **cdr-coding.** VPsim uses *cdr-coding*, while NuSim does not. Cdr-coding is a compressed representation for list elements stored in consecutive memory locations. It

requires a bit to indicate if the next location is the *car* of the next element. Cdr-coding is eliminated because its complexity has greatly complicated the microcode of the VLSI-PLM while contributing little to the overall performance [TD87].

- **instruction fetch.** NuSim does instruction fetch on demand, and accounts time for all fetches. VPsim does prefetching, which does not charge time for all fetches, but may spend time to fetch unnecessarily.

- **memory system.** NuSim has a cache/memory system with realistic values for memory access time. It accounts time for cache misses and block transfers from memory. VPsim has single (processor) cycle memory.

- **Prolog builtins.** VPsim treats some Prolog *builtins* (language predefined routines) as external functions, and ships data outside the VLSI-PLM processor for processing by the host. A varying amount of time is charged for the data shipment (3 to 10 cycles), but no time is charged for executing the external function. VPsim also implements some Prolog builtins in the library using VLSI-PLM assembly code. NuSim, on the other hand, executes all Prolog builtins inside the processor, and charges time for them as normal instructions. In NuSim, all builtins are written in C code.

6.4 The Validation of NuSim

In this section, we will compare the performance results of NuSim to those of VPsim to see how closely NuSim simulates a VLSI-PLM processor. Many benchmarks were run on both NuSim and VPsim, and their execution outputs were compared for functional correctness. A group of benchmarks was chosen for closer timing evaluation. These benchmarks differ widely in static code size and dynamic memory usage and execution time.

We have identified a number of measurements for comparison. They are: static code size, cycle count, simulation cost, operation count, and memory access count. Each type of measurement provides a different perspective of the simulation results, helping to understand the similarity and differences between the two simulators and at the same time validating the results of NuSim.

Table 6.1: Benchmark Code Sizes and Descriptions

Benchmark	lines of code NS	VP	NS/VP	Description
bintree	181	198	0.91	build a 6-node binary tree
chat	8018	8446	0.95	natural language parser
ckt4	468	370	1.27	circuit design for a mux2
compiler	11409	12488	0.91	PLM compiler (compiling bintree)
hanoi	91	82	1.11	towers of hanoi for 8 disks
mumath	262	251	1.04	Hofstadter's mumath problem for *muiiu*
nrevl	164	109	1.50	naive reverse a 30-element list
palin25	290	259	1.12	palindrome for a 25-character string
puzzle	1158	1049	1.10	solve a puzzle
qs4	249	163	1.53	quicksort on 50 numbers
qs4_meta	487	397	1.23	Prolog meta interpreter running qs4
queens6	283	294	0.96	6-queens problem
query	520	520	1.00	search a simple database
reducer	2017	2020	1.00	a graph reducer for *T-combinators*
sdda	1663	1636	1.02	static data dependency analysis
tak	69	77	0.90	solves a recursively defined function
conl	50	46	1.09	concatenation of 3- and 2-element lists
con6	53	48	1.10	pairwise partition of a 5-element list
fibo	71	69	1.03	compute 5th fibonacci number

6.4.1 Static Code Size

Table 6.1 shows the descriptions and the static code sizes (in number of lines) for the same benchmark compiled using different options for execution under NuSim (NS) and VPsim (VP). The three smallest benchmarks (conl, con6, and fibo) are listed separately at the bottom. The ratios *NS/VP* show that static NS code and VP code are for the most part well within 10% of one another. The ones that show big variances are due to the lack of cdr-coding in NuSim, which requires two instructions to build an element (car and cdr) of a list. For example, *nrev1* builds a list of 30 elements before reversing it and *qs4* builds a list of 50 elements before sorting it.

6.4.2 Cycle Count (Simulated Time)

The *simulated cycles* columns of *Table 6.2* show the cycle count of VPsim and the ratio of NuSim/VPsim cycles, respectively. The *hit ratio* column shows the cache performance for NuSim configured to a 4-way associative, 64K byte cache with a block size

of 16 bytes. From these columns, we observe that:

- Simulated time of NuSim is within 10% of that of VPsim (*NS/VP simulated cycles* are approximately 1.00) for most large programs (chat, compiler, queens6, reducer, and tak). One exception is ckt4, which is 14% slower under NuSim. This is due to differences in the implementation of instruction fetch. This will be explained in section 6.4.5.
- NuSim cycle count is significantly worse than VPsim in the small programs due to low hit ratio (cache cold start). For example, con1, con6, and fibo have the lowest hit ratios among the benchmarks, measuring at 88.7%, 95.7%, and 95.6%, respectively.
- For two programs (puzzle and query), the simulated time on NuSim is much less than that on VPsim. This is because these programs make extensive use of indexing on a term or a constant, multiplication, and division. VPsim performs these operations quite inefficiently (e.g., linear search for index term and shift-and-add for multiplication).
- Non-cdr coded lists also contribute to the slight degradation in performance for small programs such as *nrev1* (which has a decent hit ratio of 98.3%).

6.4.3 Simulation Cost

Although the time that the simulators require to run on the host is largely independent of the correctness of the results, it is interesting to compare simulation cost of the two simulators because they simulate at two different levels and follow different simulation methodologies.

The following explanations refer to Table 6.2:

- Column *VP host run time* provides the the time taken to run the simulator on the host in seconds, and column *NS/VP host run time* provides the NuSim to VPsim ratio. These numbers are obtained from running simulations on a SUN 3/60 with 16MB of memory. These values give a feel for the response time of the simulators, ranging from .5 sec to 5920 secs (or 1.64 hours).
- The *simulation cost* columns are provided as the ratio of cycle count (discussed in section 6.4.2) to host run time, assuming 100ns cycle time for the NuSim processor

Table 6.2: Cycle Count and Simulation Time

Benchmarks	simulated cycles VP	simulated cycles NS/VP	hit ratio NS	host run time (sec) VP	host run time (sec) NS/VP	simulation cost VP	simulation cost NS/VP
bintree	9875	1.30	97.8	3.5	1.43	3544	1.10
chat	6911008	1.09	99.9	1315.9	1.01	1904	0.92
ckt4	1109071	1.14	99.9	165.0	1.00	1488	0.87
compiler	2208006	0.99	99.5	529.5	0.87	2398	0.87
hanoi	78884	1.50	99.9	21.4	1.17	2713	0.78
mumath	96907	1.26	99.8	26.2	0.92	2704	0.73
nrevl	21192	1.38	98.3	6.1	1.31	2878	0.95
palin25	25026	1.08	98.6	7.4	1.08	2957	1.00
puzzle	39456475	0.67	99.9	5920.2	0.43	1500	0.65
qs4	43190	0.98	98.9	11.9	0.92	2755	0.94
qs4_meta	348051	1.17	98.9	113.6	0.65	3264	0.56
queens6	808380	1.06	100.0	125.7	1.03	1555	0.97
query	385559	0.54	99.8	61.6	0.45	1598	0.84
reducer	2543554	1.07	99.5	439.8	1.11	1729	1.04
sdda	85382	1.14	98.5	28.0	0.93	3279	0.82
tak	9398259	0.96	99.2	2461.5	0.62	2619	0.65
con1	256	2.96	88.7	0.5	6.00	19531	2.03
con6	1307	1.52	95.7	0.7	4.29	5356	2.82
fibo	2225	1.44	95.6	1.2	2.50	5393	1.73

and the VLSI-PLM chip. The VP simulation cost represents a slow down factor. For example, a value such as 2000 in these columns means that it took 2000 seconds of the SUN 3/60 time to simulate 1 second of the VLSI-PLM.

The worst numbers in the simulation cost column appear in the three smallest benchmarks con1, con6, and fibo. This is due to the initial overhead of starting up the simulators. Also in the three smallest benchmarks, the simulation cost of NuSim is much higher than VPsim (1.73 to 2.82 times worse). This is because NuSim takes more time to startup, being a multiprocessor simulator and having to assemble the benchmark into assembly code. For the larger benchmarks, NuSim is more efficient than VPsim. Excluding the three smallest benchmarks, the average simulation costs of NuSim and VPsim are 2079 and 2430, respectively. Thus NuSim is 14% more efficient.

- Even though NuSim simulates the VLSI-PLM at a slightly higher level than the register-transfer level of VPsim, it is not that much more efficient because VPsim

microcode is "flat" while NuSim C-routines are hierarchically structured. The cost of structured code depends on the efficiency of the code generated by the C compiler for subroutine calls and returns.

Simulation of the VLSI-PLM on a SUN 3/60 is more than 2000 times slower than actual execution on a VLSI-PLM because of the following reasons:

- VLSI-PLM code is represented internally as ASCII strings, which require longer processing time.
- Data and control transfers (the microcode) are processed sequentially. In a real machine, it would be done in parallel. The VLSI-PLM has a two stage pipeline, with the data unit and microsequencer executing in parallel. The VLSI-PLM data unit is also capable of doing 8 simultaneous transfers in one cycle.
- The host processor is less powerful than the target processor for symbolic computation and the host memory access time is slower than the target memory access time. The SUN 3/60 that we use has a 20MHz MC68020 and 16MB of main memory (300ns access time). There is no cache. The VLSI-PLM is a complex processor with tag processing capability.
- The code generated by the C compiler affects the execution time of the host. For example, inefficient subroutine calls and returns penalize the hierarchical structure of NuSim C code.
- The presence of extensive instrumentation code in the simulators for extracting performance results slows down execution on the host.
- The operating system characteristic of the host can greatly affect performance. The SUN 3/60 runs 4.3 BSD Unix^{TM} and virtual memory. The CPU accesses a shared file server connected via the Ethernet, and thus page faults are very expensive.

The factors above blend together in the real uniprocessor system and it is difficult to measure them separately. This is the reason why a simulator is needed for experimentation with individual system parameters. For simulating a multiprocessor configuration, the event driven approach of NuSim may be accelerated by use of a faster uniprocessor, or a multiprocessor host, as demonstrated by [Wil87b, Jon86]. For the greatest efficiency in

Table 6.3: Logical Inference Count

Benchmark	NS calls	NS escapes	NS KLIPS	VP calls	VP escapes	VP KLIPS	VP/NS KLIPS
bintree	77	151	177	128	101	232	0.76
chat	66905	60	89	66911	55	97	0.92
ckt4	3544	916	35	4458	3	40	0.87
compiler	15113	7186	102	20886	2539	106	0.96
hanoi	767	765	129	1022	511	194	0.67
mumath	1211	82	106	1221	73	134	0.79
nrev1	497	2	171	497	3	236	0.72
palin25	228	97	121	323	3	130	0.93
puzzle	19796	6018	10	21800	4015	7	1.50
qs4	381	231	144	610	3	142	1.02
qs4_meta	2694	720	84	3795	3	109	0.77
queens6	3207	6130	109	9337	9	116	0.94
query	703	2835	170	2878	661	92	1.85
reducer	15091	6305	79	18815	2491	84	0.94
sdda	552	408	99	715	249	113	0.87
tak	63609	111317	195	174924	3	186	1.05
con1	4	2	79	4	3	273	0.29
con6	6	30	181	6	31	283	0.64
fibo	15	23	118	36	3	175	0.68

simulation, a *direct execution* approach such as the one proposed by Fujimoto [FC88] may be used, where the benchmark is compiled into code directly executable by the host. Instrumentation counters are inserted by the compiler into the code to measure performance for the target machine.

6.4.4 Operation Count

In Prolog, the metric *Kilo Logical Inferences Per Second* (KLIPS) is often used for measuring the performance of Prolog engines. In this dissertation, a logical inference is defined as a Prolog function call, which includes the VLSI-PLM instructions *call*, *execute*, and *escape* for Prolog builtins. This metric is quite inaccurate since the time required for a logical inference can vary greatly, depending on the primitive operations required. The amount of work done by a Prolog function call depends on the *number* and *type* of arguments in Prolog. For parallel execution, the KLIPS measurement has even less significance. Multiprocessors may do more work but do not necessarily achieve the final

result any faster, if the additional computations do not contribute directly to the result.

Table 6.3 shows the number of normal calls (and executes) and Prolog builtin invocations (or escapes). Since VPsim does calls to library routines for some of the builtins, it has a much higher calls count and fewer escape count than NuSim. In order for KLIPS to be a useful measure, the condition $NS_{call} + NS_{escape} \approx VP_{call} + VP_{escape}$ should hold true. The following results show that this condition does not hold, due to the implementation variations of NuSim and VPsim (described in section 6.3.2).

Each of the KLIPS columns is calculated by

$$\frac{calls + escapes}{cycles} * 10000$$

where *cycles* is obtained from Table 6.2. The unit for *calls* and *escapes* is the logical inference. The constant factor of 10000 comes from the KLIPS unit conversion:

$$1\ KLIP = \frac{10^9\ nsec}{1\ sec} * \frac{1\ cycle}{100\ nsec} * \frac{1\ K}{1000}$$

The *NS KLIPS* and *VP KLIPS* columns differ widely, showing once again the problem with this metric. For comparison purpose, the timing information in table 6.2 is much more useful than this metric.

6.4.5 Memory Accesses

Table 6.4 compares the number of memory accesses made in running the simulations on NuSim and the VPsim. The *VP total references* column gives the total number of memory accesses, which ranges from about 100 to over 11 million. The other columns show the breakdown of references into instruction fetches, data reads, and data writes.
The following can be observed:

- The total references for most programs under the two simulators are within 20% of each other. The biggest variations are for con1 (2.11), con6 (1.58), and nrev1 (1.51). The variations are perfect examples of worst case performance without cdr-coding (in NuSim), which would require more instruction fetches, reads and writes. For the larger benchmarks, cdr-coding makes little difference.
- For the most part, NuSim requires more instruction fetches than VPsim. This is because NuSim instructions are less compact than VPsim. They are encoded in word

Table 6.4: Memory References

Benchmark	total references VP	$\frac{NS}{VP}$	instruction fetches VP	$\frac{NS}{VP}$	data reads VP	$\frac{NS}{VP}$	data writes VP	$\frac{NS}{VP}$
bintree	5601	1.19	2527	1.03	1568	1.80	1506	0.82
chat	3695155	1.16	1376937	1.50	1158506	0.95	1159712	0.97
ckt4	547619	1.25	149409	1.95	249946	0.97	148264	1.02
compiler	1259778	1.07	470464	1.18	426110	1.06	363204	0.93
hanoi	51811	1.38	21441	1.65	13776	1.26	16594	1.12
mumath	53052	1.29	18258	1.78	18639	1.03	16155	1.03
nrev1	8473	1.51	4812	1.97	2017	0.81	1644	1.06
palin25	12759	1.10	5695	1.31	4114	0.89	2950	0.99
puzzle	11600446	0.81	771251	1.59	9498654	0.72	1330541	0.99
qs4	24302	0.93	11141	1.04	5509	0.87	7652	0.79
qs4_meta	197469	1.13	70542	1.42	61671	0.97	65256	0.96
queens6	504104	1.09	212691	1.24	172009	0.98	119404	1.00
query	102771	1.01	60780	0.92	27513	1.30	14478	0.83
reducer	1367058	1.14	462255	1.46	507144	0.99	397659	0.97
sdda	48313	1.13	17831	1.33	16752	1.08	13730	0.95
tak	5979238	0.83	3291760	0.66	1033643	1.18	1653835	0.96
con1	94	2.11	55	2.07	17	2.94	22	1.55
con6	499	1.58	163	1.84	170	1.86	166	1.04
fibo	1207	1.10	648	1.13	215	1.24	344	0.95

streams, with the opcode and each operand taking up one 32-bit word. VPsim has the code stored in string tables, but the microcode generates prefetch signals to simulate an encoding of 8-bit opcode and 32-bit arguments. Furthermore, NuSim fetches instructions on demand, while VPsim does prefetching.

- The instruction fetch ratios show that the word-encoding of NuSim require more fetches, as expected. However, for *tak*, NuSim fetches much less (instruction fetch ratio of 0.66) because many subtractions are done and NuSim use the builtin instruction *is/2*, while VPsim does a call to the library routine *sub/3* which require a longer sequence of simpler instructions.

- The reasons for the differences in the number of data reads include: (a) cdr-coding in VPsim require fewer reads, and (b) solving arithmetic expression in VPsim require fewer reads because VPsim has simple arithmetic instructions (*add* and *subtract*). In NuSim, arithmetic expressions are put in structures and passed to the *is/2* builtin. For the most part, the number of data writes for both VPsim and NuSim are the

same.

6.5 Chapter Summary

In this chapter, we have described an approach for validating the simulator of a multiprocessor system. The processor and the memory module of a multiprocessor simulator (NuSim) has been validated by comparing it with a previously validated uniprocessor simulator (VPsim). Benchmarks of various sizes were executed sequentially on both simulators, and different performance measurements were evaluated and compared against one another.

Because the simulation result is a composite result of many factors, a number of measurements were used for comparison to obtain different perspectives on performance and to understand the reasons of the variations. The chosen measurements were: code size, cycle count, simulation cost, operation count, and memory access count. The different measurements indicate that the variations are significant only for the small benchmarks, where startup time and slight model differences are a big percentage of total execution time. For large programs, NuSim is within 10% of the VLSI-PLM timing. Perhaps more importantly, all variations can be accounted for. It can be concluded that NuSim is representative of a VLSI-PLM in a multiprocessor system. With NuSim, the performance of memory management for a shared memory multiprocessor can be evaluated.

Chapter 7

ELPS Simulation Experiments and Results

This chapter presents the experiments done using the multiprocessor simulator and the corresponding results. The first part is an evaluation of the potential effect on performance for each of the variations to the sequential model. The second part provides the overall performance of the ELPS memory management scheme described in chapter 4.

7.1 Sequential Execution Performance

The ELPS memory management scheme uses discontiguous pages of memory to form a logically continuous stack. In the most general case where the addresses of the pages may be in arbitrary order, address comparison of two data objects in the stack to determine their relative order is not possible. Section 4.3.3 proposed modifications to the WAM to eliminate the need for address comparison. This section presents the simulation results from each of these modifications, to evaluate their effects on sequential execution performance. For the simulation runs, the cache is configured to a size of 16K words and is 4-way associative, with a block size of 16 bytes, LRU replacement, and write back policy.

7.1.1 Split Environment and Choice Point Stacks

The WAM local stack is a *combined stack* model, which contains both choice points and environment. As described in Tick's dissertation [Tic87], it can be split into

two stacks (*split stack* model), one for each type of control structure. The choice point stack is a true stack, where the *topmost* choice point is always the *current* choice point. On the other hand, the *current* environment is not necessarily the one at the top of the environment stack. Splitting the two stacks has the advantage of greater access locality for choice points and ability to reclaim space of the choice points which may be trapped deeply below an environment, or of environments trapped below a choice point. *Figure 7.1* shows a pathological example for which a split stack is much better. On the negative side, the split stack model has the complexity of maintaining an extra stack pointer.

Table 7.1 shows the space and time performance of split stack versus combined stack. As shown in the *Cycles* ratio split/combined (s/c) column, the split stack model execution times is within 3% those of the combined stack model. With respect to space usage, the maximum space used by heap and trail are identical for both models. For the environment and choicepoint stacks, the maximum space usage of the split stack model is mostly the same as that of the combined stack model, with some programs having a maximum usage of 90% of the combined stack model (column *envir+choicept (s/c)*). The overall effect of these programs is no more than 4% (column *total (s/c)*). The only exception to this trend is the benchmark *sdda*, whose maximum (environment+choicepoint) usage of split stack is only 56% of that of combined stack, with the overall space usage of split stack reaching only 73% of that of combined stack. The reason is that *sdda* has a section of code which is very similar to the example shown in *figure 7.1*, where the choicepoints and environments trap one another, preventing space from being reclaimed.

Table 7.2 shows the memory statistics of split stack versus combined stack, which include cache hit ratio, bus utilization, memory references (to/from cache memory), reads from main memory (bus reads), and writes to main memory (bus writes). For each of these measurements, the value for combined stack is presented together with the ratio of the corresponding value of split stack over combined stack (s/c). The hit ratio is almost identical for both models. The split stack model results in an average of 3% more memory accesses. This is the effect of managing an extra stack pointer. The bus utilization goes down slightly for split stack. Since there is a greater degree of locality for the separate choicepoint stack, the number of bus reads and bus writes also decrease slightly. The zero value in the bus writes column indicate that the writeback cache is big enough such that no flushing to memory is needed. Correspondingly, the split/combined ratio column is left empty.

```
% Pathological example to show the advantage of split stacks
% The program alternately creates choicepoints and environments
% which trap one another, and doesn't reclaim space in
% combined stack.  (Courtesy of Peter Van Roy)

?- p(100).

p(0) :- !.
p(N) :- d, !,             % create env before a cut to trap choicept for p()
        N1 is N-1,
        ch,               % create choicept to trap env
        p(N1).
p(_).

d.
ch.
ch.
```

Figure 7.1: Disadvantage of Combined Environment/Choicepoint Stack

In the program above, a choicepoint for p() is created. In the clause p(N), an environment is laid down before procedure d is called to store return address. When the cut (!) is reached, the top choicepoint pointer is moved down to a previous choicepoint, deallocating choicepoint for p(). However, since the choicepoint is below the environment of p(N), this deallocated space cannot be reclaimed. When procedure ch is called, a choicepoint is created for ch. When p(N1) is reached, environment for p(N) is deallocated, but space cannot be reclaimed because it is below choicepoint for ch. In the split stack model, no such trapping can occur.

Table 7.1: Split vs. Combined Environment/Choicepoint Stack

	Cycles		Maximum Space Usage				
			heap	trail	envir+choicept		total
Benchmark	combined	split/comb.			combined	s/c	s/c
bintree	13508	1.00	75	5	426	1.00	1.00
boyer	46507026	1.01	544023	95274	634	1.02	1.00
browse	2565514	1.01	665	114	399	1.00	1.00
chat	7673061	1.01	1518	414	2592	0.93	0.96
ckt4	1269879	1.02	141	71	377	1.02	1.02
compiler	2259298	1.00	4769	791	2021	0.89	0.97
qs4	43439	1.00	658	107	85	1.00	1.00
qs4_meta	428072	1.01	4460	1223	7504	1.00	1.00
query	208743	1.03	12	8	58	1.00	1.00
reducer	2788988	1.01	29052	8032	2395	0.90	0.99
sdda	102395	0.99	938	269	1916	0.56	0.73
tak	9384597	0.99	238530	0	200	1.00	1.00
tp3	3369301	1.01	29223	8300	3751	1.00	1.00
average		1.01				0.95	0.98

Table 7.2: Memory Statistics for Split vs. Combined Stack

	hit ratio		bus util.		mem. references		bus reads		bus writes	
Benchmark	comb.	s/c	com.	s/c	comb.	s/c	comb.	s/c	comb.	s/c
bintree	96.8	1.00	12.9	1.00	6730	1.02	868	1.00	0	
boyer	99.3	1.00	4.7	0.98	25672217	1.04	690916	1.00	630364	1.00
browse	100.0	1.00	0.2	1.00	1558652	1.03	2680	1.00	0	
chat	99.9	1.00	0.6	1.00	4330329	1.04	22048	0.96	3004	0.85
ckt4	99.9	1.00	0.2	1.00	683432	1.05	1472	1.01	0	
compiler	99.3	1.00	3.8	0.92	1353377	1.03	38664	0.94	6528	0.83
qs4	98.5	1.00	6.2	1.02	22412	1.02	1356	1.00	0	
qs4_meta	98.4	1.00	6.8	0.99	225354	1.04	14140	1.00	504	1.01
queens6	100.0	1.00	1.1	1.00	551963	1.03	860	1.00	0	
query	99.8	1.00	4.2	0.98	104076	1.05	800	1.00	0	
reducer	99.3	1.00	4.3	0.98	1568647	1.04	43108	1.00	26200	0.99
sdda	97.7	1.01	9.7	0.84	54616	1.03	4944	0.83	0	
tak	98.8	1.00	8.1	1.00	4945568	1.00	239244	1.00	222740	1.00
tp3	99.4	1.00	4.0	0.97	1915884	1.03	48888	1.00	29144	1.00
average		1.00		0.98		1.03		0.98		0.95

From these results, we can conclude that *it is clearly advantageous to split the environment and choicepoint stack.* We thus make use of split stack for the next two experiments.

7.1.2 Always Trail

In the original WAM model, checks are performed to decide whether it is really necessary to trail a binding. The binding of a variable needs to be trailed only if the variable is created before the current choicepoint. The check is performed by comparing the variable's address with the top choicepoint pointer (B register), if the variable is a permanent variable on the stack, or with the heap backtrack pointer (HB register), if the variable is on the heap. In the split stack model, the stack variable must be compared with the top of the environment stack at the time that the choicepoint is created on the choicepoint stack. This environment stack pointer is saved in the choicepoint. The question of whether or not to perform the trail check is a tradeoff between type of memory access (read or write) and space. The trail checks require more memory reads and more time for the comparison, but do fewer writes and thus use less space if trailing is not necessary.

Table 7.3 shows a comparison of selective trail (which performs trail checks) and always trail (which does no checking). The *Cycles* always trail/selective trail (a/s) ratio column shows that both models take about the same amount of time to execute. The maximum space usage for heap, environment, and choicepoint are identical for both models. However, the trail space usage columns shows that the lack of trail checks can lead to explosion in trail space usage. Program ***tak*** is an extreme case which does not trail at all with trail checks, and require almost 100K words of trail space without trail checks. The overall space effect (column ***total (always/selective)***) can be as high as 62% more space for program ***boyer***. Perhaps most importantly, since trail space is often a very small percentage of overall space usage, such explosion in space usage would result in overflowing the small space reserved for the trail stack.

Table 7.4 shows the effect that always trailing has on the memory system. Always trailing has little to no effect on the hit ratio and the number of memory references. A trail check requires a read, whereas a trail operation requires a write. The excessive number of writes for always trailing causes the block in the cache to be flushed to memory and later brought back into the cache. This explains the sharp increase in bus reads and bus writes

Table 7.3: Always Trail vs. Selective Trail (split stacks)

	Cycles		Maximum Space Usage						
			heap	env	chpt	trail			total
	selective	a/s				sel.	always	a/s	a/s
bintree	13560	1.00	75	81	345	5	51	10.20	1.09
boyer	47059439	1.01	544023	394	255	95274	494444	5.19	1.62
browse	2585430	0.98	665	85	315	114	470	4.12	1.30
chat	7786363	0.97	1518	1062	1350	414	577	1.39	1.04
ckt4	1299223	0.95	141	41	345	71	102	1.44	1.05
compiler	2270210	1.00	4769	715	1080	791	2007	2.54	1.17
qs4	43504	1.00	658	70	15	107	431	4.03	1.38
qs4_meta	433001	0.99	4460	4129	3375	1223	2544	2.08	1.10
queens6	867112	0.99	43	123	195	13	22	1.69	1.02
query	214477	0.96	12	28	30	8	17	2.13	1.12
reducer	2824152	0.99	29052	1100	1050	8032	15882	1.98	1.20
sdda	101007	0.99	938	557	510	269	596	2.22	1.14
tak	9337079	1.04	238530	185	15	0	95413		1.40
tp3	3406891	1.01	29223	2521	1230	8300	29619	3.57	1.52
average		0.99						3.27	1.23

Table 7.4: Memory Statistics for Always vs. Selective Trail (split stacks)

	hit ratio		bus util.		mem. references		bus reads		bus writes	
Benchmark	sel.	a/s	sel.	a/s	sel.	a/s	sel.	a/s	sel.	a/s
bintree	96.8	1.00	12.9	1.05	6855	1.00	868	1.05	0	
boyer	99.4	1.00	4.6	1.59	26618704	0.99	690984	1.60	630532	1.64
browse	100.0	1.00	0.2	1.00	1603985	1.00	2684	1.13	0	
chat	99.9	1.00	0.6	1.00	4494333	1.00	21132	1.01	2548	1.04
ckt4	99.9	1.00	0.2	1.00	717937	0.98	1484	1.02	0	
compiler	99.3	1.00	3.5	1.23	1392346	1.01	36488	1.19	5420	1.51
qs4	98.5	1.00	6.3	1.24	22766	1.00	1360	1.24	0	
qs4_meta	98.5	1.00	6.7	1.10	233582	0.99	14144	1.09	508	0.97
queens6	100.0	1.00	1.1	1.00	568335	1.02	864	1.01	0	
query	99.8	1.00	4.1	1.05	109195	1.00	804	1.01	0	
reducer	99.3	1.00	4.2	1.24	1625630	0.99	42904	1.18	26044	1.30
sdda	98.2	1.00	8.1	1.10	56368	0.99	4096	1.08	0	
tak	98.8	0.99	8.1	1.36	4961472	1.02	239288	1.40	222776	1.43
tp3	99.4	1.00	3.9	1.54	1975138	1.00	48960	1.47	29164	1.73
average		1.00		1.18		1.00		1.18		1.37

(and correspondingly, bus utilization).

Under static partitioning, the potential space explosion can cause serious overflow problems if it is not handled properly. ELPS should be able to handle this quite efficiently. However, the extra writes can heavily tax the multiprocessor memory system, resulting in performance degradation. Therefore, ***always trailing is not a good idea in general.*** However, it should be noted that always trailing may be a good idea in a system where:

- memory write is as fast as memory read by employing write buffering.
- there is hardware support for stack write optimization; that is, pushing onto the stack does not require reading the block from memory into cache.
- frequent garbage collection is performed; the trail space can be quickly reclaimed, and thus the space explosion and excess writes will become less problematic.

7.1.3 Put Permanent Variables on Heap

A variable is referred to as *permanent* if it needs to be retained across the goals in the body of a clause. In the WAM model, permanent variables are put in the stack, together with the environment. Since the tail recursion optimization discards the environment before the last call in a clause, the permanent variables are copied onto the heap after this last call using the `put_unsafe_value` instruction. The WAM model requires that a variable must be checked to see whether it resides on the heap or on the stack for proper trail operation. An alternative is to immediately place the unbound variable on the heap, with the stack variable pointing to it. The dereference operation will always follow the pointer to the variable on the heap, thus it is not necessary to check for a variable on the stack. Putting all permanent variables on heap simplifies the trail check and eliminates the need for the `put_unsafe_value` instruction. On the negative side, it may require more heap space and longer access time due to the extra level of indirection.

Table 7.5 shows the comparison between the standard WAM model and the variation of permanent variable on heap. The *Cycles* on heap/on stack (hp/stk) column shows little difference in execution times. The maximum space usage for environment and choicepoint stacks are identical under both models. The *heap* ratio column shows an average increase of 18% for putting permanent variables on heap. Several programs exhibit a greater than 20% increase in heap space usage. For these programs, `put_unsafe_value` statically

Table 7.5: Permanent Variables on Heap vs. on Stack (split stacks)

	Cycles		Maximum Space Usage						
			envir	chpt	heap		trail		total
Benchmark	on stack	$\frac{hp}{stk}$			on stk	$\frac{hp}{stk}$	on stk	$\frac{hp}{stk}$	$\frac{hp}{stk}$
bintree	13560	1.00	81	345	75	1.09	5	1.00	1.01
boyer	47059439	1.01	394	255	544023	1.14	95274	1.00	1.12
browse	2585430	1.00	85	315	665	1.05	114	1.04	1.03
chat	7786363	1.01	1062	1350	1518	1.17	414	0.95	1.05
ckt4	1299223	1.00	41	345	141	1.07	71	1.00	1.02
compiler	2270210	1.01	715	1080	4769	1.10	791	1.20	1.09
qs4	43504	1.02	70	15	658	1.23	107	1.00	1.18
qs4_meta	433001	1.02	4129	3375	4460	1.30	1223	1.03	1.11
queens6	867112	1.01	123	195	43	1.16	13	1.00	1.02
query	214477	1.00	28	30	12	1.42	8	0.75	1.04
reducer	2824152	1.00	1100	1050	29052	1.06	8032	0.99	1.04
sdda	101007	1.01	557	510	938	1.11	269	1.01	1.05
tak	9337079	1.03	185	15	238530	1.20	0		1.20
tp3	3406891	1.02	2521	1230	29223	1.41	8300	1.09	1.31
average		1.01				1.18		1.00	1.09

Table 7.6: Memory Statistics for Perm. Vars. on Heap vs. on Stack (split stacks)

	hit ratio		bus util		mem. references		bus reads		bus writes	
Benchmark	stk	$\frac{hp}{stk}$	stk	$\frac{hp}{stk}$	on stack	$\frac{hp}{stk}$	stk	$\frac{hp}{stk}$	stk	$\frac{hp}{stk}$
bintree	96.8	1.00	12.9	1.01	6855	1.00	868	1.01	0	
boyer	99.4	1.00	4.6	1.11	26618704	1.01	690984	1.12	630532	1.12
browse	100.0	1.00	0.2	1.00	1603985	1.01	2684	1.01	0	
chat	99.9	1.00	0.6	1.00	4494333	1.02	21132	1.02	2548	1.14
ckt4	99.9	1.00	0.2	1.00	717937	1.00	1484	1.01	0	
compiler	99.3	1.00	3.5	1.09	1392346	1.01	36488	1.09	5420	1.29
qs4	98.5	1.00	6.3	1.08	22766	1.01	1360	1.11	0	
qs4_meta	98.5	1.00	6.7	1.07	233582	1.02	14144	1.10	508	1.13
queens6	100.0	1.00	1.1	1.00	568335	1.01	864	1.01	0	
query	99.8	1.00	4.1	1.00	109195	1.00	804	1.00	0	
reducer	99.3	1.00	4.2	1.05	1625630	1.00	42904	1.04	26044	1.06
sdda	98.2	1.00	8.1	1.02	56368	1.00	4096	1.03	0	
tak	98.8	1.00	8.1	1.17	4961472	1.02	239288	1.20	222776	1.21
tp3	99.4	1.00	3.9	1.31	1975138	1.01	48960	1.28	29164	1.44
average		1.00		1.06		1.01		1.07		1.20

occurs often in recursive procedures. However, the permanent variable does not get created on the heap if before the put_unsafe_value instruction is reached, failure occurs or the variable gets bound.

The trail usage on the average is the same in both models. The fluctuations in the trail column are due to three reasons:

1. `put_unsafe_value` performs a trail operation when it copies a variable from the stack onto the heap, while the variable-on-heap model does not trail. Thus, if the variable is bound before the put_unsafe_value instruction is reached, both models will do the same amount of trailing.

2. If the variable is still unbound, the variable-on-heap model does less trailing.

3. The recovery of the heap is not optimal at the time of `trust`, `trust_me_else`, cut and cutd. The heap backtrack (HB) pointer is reset to the saved heap (H) pointer in the choicepoint being cut. Optimally, the heap backtrack pointer should be reset to the saved heap pointer of the choicepoint below the choicepoint being cut, if it exists, or to the heap base, if there is no current choicepoint. This optimal reset costs extra time for the check and possibly an additional memory read. A simple solution to this is to create a dummy choice point at the bottom of the choice point stack for use as a sentinel.

Overall, the total space usage for putting permanent variables on the heap requires 9% more space on average.

Table 7.6 shows the memory performance of the variable-on-heap model, as compared with the original WAM model. Once again, the hit ratio and the number of memory references are not affected by the model variation. However, the change in heap and trail usage causes a corresponding change in average bus utilization (6% increase), bus reads (7% increase), and bus writes (20% increase).

In summary, putting variables on the heap results in 18% increase in heap space usage (9% overall) and 6% increase in bus utilization, but does not cost any additional cycles. Furthermore, it greatly simplifies the trail check operation and eliminates the need to copy variables from the stack to the heap when the environment is discarded. Therefore, we can conclude that *putting variables on the heap is an acceptable variation to the original WAM model.*

7.2 Parallel Execution and ELPS Performance

Seven Prolog benchmark programs are chosen for our study of ELPS in parallel execution. These programs exhibit a variety of parallelism characteristics and memory usage. *Table 7.7* lists the benchmarks, their static code sizes, and a brief explanation for each. Prolog programs are annotated for parallel execution, compiled into assembly code, and loaded into the simulator.

Table 7.7: Benchmark Code Sizes and Descriptions

Benchmark	lines of Prolog code	Description
boyer	396	Boyer-Moore theorem prover
chat	1196	natural language parser
ckt4	48	circuit design for a 2-to-1 mux
compiler	2271	a Prolog compiler
qsd200	18	quicksort on list of 200 data items
query	71	multiple queries of a simple database
tp3	763	Overbeek's theorem prover

Qsd200 is a version of quicksort using a data structure called the difference list. This allows the two subpartitions of a list to be sorted in parallel and afterwards linked together in constant time.

7.2.1 Execution Time Overhead

Table 7.8 shows the execution times of the programs for three configurations: static partitioning, ELPS with hardware support, and ELPS with software only. The overhead percentage is computed by $(\frac{ELPS\ time}{static\ partitioning\ time} - 1) \times 100$. To study the overhead of ELPS, the number of tasks is set to a maximum of 64 for all configurations. ELPS page size is set at 4K words so that overflow does not occur in most programs[1]. Furthermore, the multiprocessor system is configured to single cycle memory to factor out the cache effects. The cache effects will be considered in section 7.2.3. As explained in section 5.9, no time is charged for checks with hardware support, and two cycles are charged for each overflow check with software only. Without any overflow, the overhead of ELPS includes the extra time incurred by: (a) the checks for page overflow (if software only), and (b) the checks for variable locality (for OR-parallelism and always done in software). A variable is local

[1] Overflows occur only in boyer and tp3.

Table 7.8: Overhead of ELPS Checking and Overflow Handling

benchmark	static partitioning (cycle)	ELPS with hardware support (cycle)	(% overhead)	ELPS with software only (cycle)	(% overhead)
query	34757	35969	3.5	39142	12.6
qsd200	67050	67610	0.8	75788	13.0
compiler	1088084	1101923	1.3	1190858	9.4
ckt4	1468717	1516685	3.3	1670465	13.7
chat	2290302	2347801	2.5	2580181	12.7
tp3	3213414	3254666	1.3	3449535	7.3
boyer	51370794	52008092	1.2	56096531	9.2
arith mean			2.0		11.1
geom mean			2.0		10.9

(or internal) to a task[2] if it exists on the heap stack of that task. The linked list of pages forming the heap stack may need to be traversed to determine if the variable lies in one of the pages.

With hardware support for overflow checking, the overhead for all programs ranges from 0.8% to 3.5% (2% on average). With software-only checking, the overhead is quite a bit higher, ranging from 7.3% to 13.7% (11% on average). Thus hardware support provides an average of 9% improvement in total execution time over software-only checking.

Table 7.9 shows a breakdown of ELPS behavior. In this table, the page size is set to 1K for greater overflow frequency. The first column shows the average number of cycles between checks. The average over all programs, except query, is 87 cycles between checks. Query requires very infrequent overflow checks (one every 4831 cycles) because it spends most of the time reading a database and writes very infrequently to the stacks. The next two columns show the number of overflows and the average time required to handle an overflow. The number of overflows depend greatly on the page size. An advantage of ELPS is very fast overflow handling which is in tens of cycles.

The new page request percentages (number of new page requests/number of overflows) indicate the degree of stack pointer movement across page boundaries. When a page overflow occurs for the first time, a new page is obtained. When the stack underflows to the previous page, the current page is retained (lazy deallocation) so that subsequent overflows do not require new pages. Boyer and chat are examples of opposite extreme behaviors. The

[2] Bindings to internal variables are stored in place at the specified address, while bindings to external variables are stored in the hashwindow (previously discussed in section 3.4.5).

Table 7.9: Behavior of ELPS Checking and Overflow Handling

benchmark	cycles between checks	number of overflows	avg overflow handling time	new page requests (%)	times unable to spawn*
ckt4	126	0	-	-	0
query	4831	0	-	-	23
qsd200	84	1	31	100.0	137
compiler	73	26	26	69.2	0
tp3	80	129	15	52.7	279
chat	68	357	2	2.5	0
boyer	92	1851	23	98.3	93993

* under static partitioning

stacks in boyer primarily grow upward (98.3% new page requests), while the stacks in chat backtracks very frequently (only 2.5% new page requests). For chat, lazy deallocation is clearly advantageous. It results in an average overflow handling time of only 2 cycles.

The last column in table 7.9 shows the number of times that new tasks could not be spawned because the number of tasks is limited (to 64), with all unused space statically allocated to other tasks. This column shows the key advantage of ELPS. With ELPS, memory is efficiently distributed to keep up with the demand for a very large number of tasks. For ELPS, this unable-to-spawn column would typically be zero. While ELPS provide memory space support for a high degree of parallelism, the resulting speedup depends on the ability of the scheduler to efficiently exploit parallelism (i.e., to spawn a parallel task only when the amount of work to be done by the new task is sufficiently higher than the overhead of task creation, communication, and termination).

Fragmentation is another performance measure of ELPS. *Internal* fragmentation occurs when the space at the top of each page is insufficient to store the data object. *External* fragmentation is the amount of space unused on the section of the page beyond the top of stack pointer. Compared to a smaller page size, a larger page size will tend to have greater external fragmentation but less internal fragmentation. For the chosen set of benchmark programs, internal fragmentation is consistently very small, averaging less than 10 words per 1K word page (less than 1%). External fragmentation varies greatly from program to program. Compared with static partitioning, ELPS has slightly more internal fragmentation (none in static partitioning), but much less external fragmentation (ELPS

page size is much smaller than a static partition).

7.2.2 Parallelism Gained

A key point of ELPS is efficient sharing of the global address space to allow a very large number of tasks to be spawned. An example of the degree of parallelism gained with more tasks is shown in *figures 7.2 (a)-(d)*. Each graph captures the same period of time of the execution of *boyer*, a Prolog version of the Boyer-Moore theorem prover.

In the top two graphs, the maximum number of tasks is set to 64, with ID numbers from 0 to 63. Each horizontal segment in figure 7.2(a) shows the period of time in which a task executes in some processor. With the simulated architecture configured to 8 processors, any vertical line has at most 8 intersection points. The diagonal slope shows the start of the task creations as new tasks are spawned during execution. Task space may be reclaimed only if that task terminates. After some time, all tasks but one go into sleeping state and hang on to their task space for potential future backtracking. When all tasks are used, execution proceeds sequentially (as shown by the long horizontal line from cycle 70000 to cycle 165000 of figure 7.2(a)).

Figure 7.2(b) shows the corresponding processor utilization, with only one processor busy since time 75000. In the bottom two graphs, the maximum number of tasks is increased to 256 (task ID 0-255). Four times more tasks are now spawned, and all eight processors are kept quite busy during the specified window of time.

Over the entire execution, boyer runs 2X faster with 256 tasks than with 64 tasks. However, a maximum of 256 tasks is still insufficient. When this maximum is reached, it runs sequentially until the end. If the parallel execution pattern is extrapolated over the entire run, we can expect a 4X to 6X speed up (with 8 processors and a very large number of tasks) over the execution run with 64 tasks maximum.

ELPS allows for a very large number of tasks. The significant potential of advantage of ELPS over static partitioning can be illustrated by a specific example. Consider a 32-bit address space and a set of programs that use tens of words up to 1M words for each stack. For static partitioning, stack must be configured to the largest possible size to avoid overflow. If each stack is sized at 1M words, there can be at most $\frac{2^{32}}{2^{20}} = 2^{12} = 4096$ *stacks*. Suppose that only 20% of the stacks are near the 1M-word usage while the others are less than 1K (such unpredictable usage is often the case for parallel execution of Prolog under

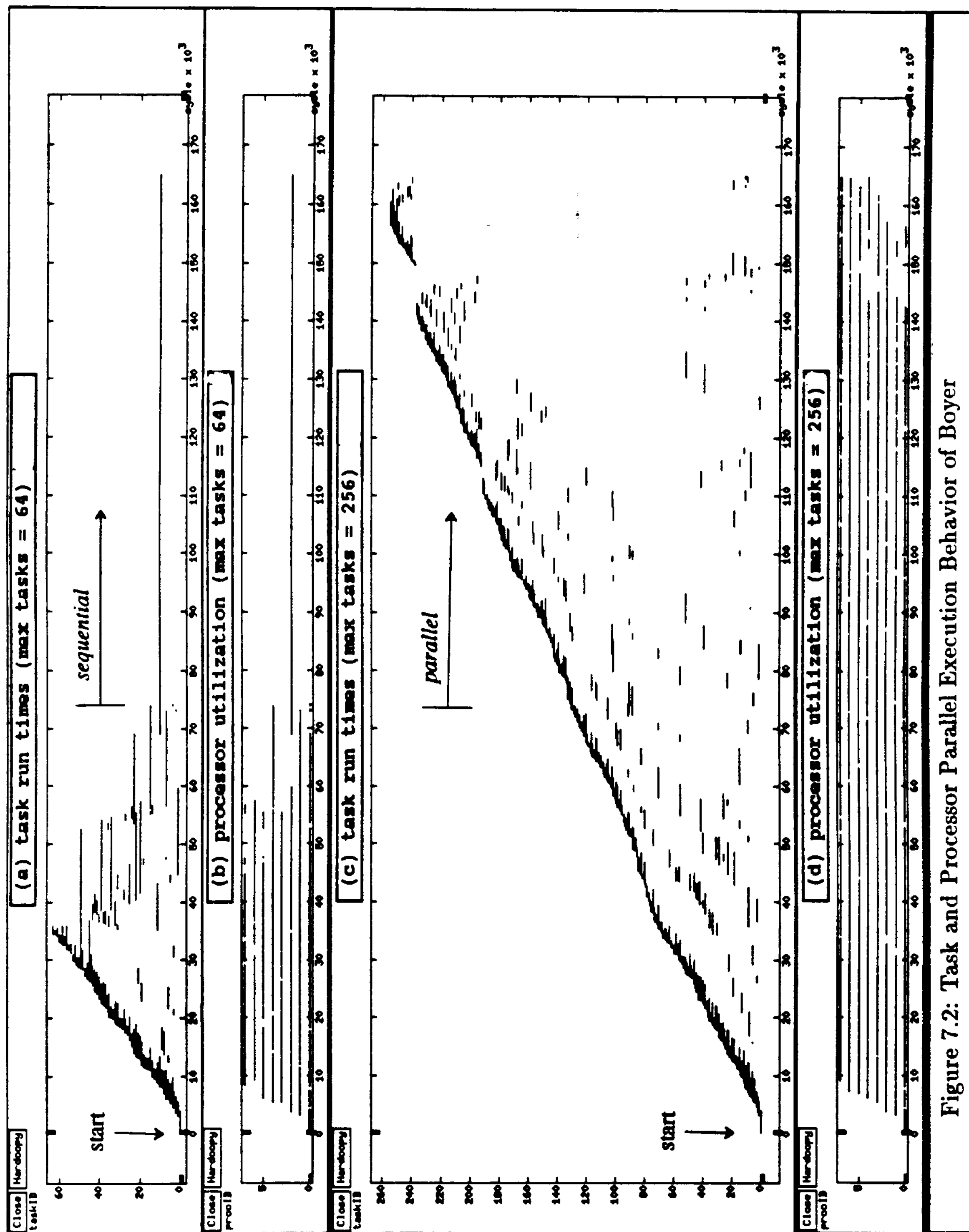


Figure 7.2: Task and Processor Parallel Execution Behavior of Boyer

Table 7.10: Effect of ELPS Page Size on Execution Time

Benchmark	static partitioning (cycle)	Overhead % of ELPS/static partitioning: no hashing for data start (page sizes in words) 512	1K	2K	4K	8K	with hashing 8K page (a)	(b)
qsd200	97048	3.6	8.0	11.4	14.5	14.5	5.4	7.5
query	63586	6.7	8.9	12.7	26.0	26.0	5.2	5.3
chat	2353059	3.2	3.4	3.5	4.9	4.9	3.7	4.7
ckt4	1554513	4.4	4.4	4.3	4.9	4.9	4.4	4.4
compiler	1261500	3.5	3.1	3.4	5.1	5.1	4.1	3.9
boyer	55100234	-	6.9	5.3	4.6	4.1	4.1	4.1
tp3	3394992	6.4	5.3	4.9	4.5	4.3	4.3	4.3
arith mean		4.6	5.7	6.5	9.2	9.1	4.5	4.9
geom mean		4.4	5.3	5.7	7.1	7.0	4.4	4.8

(a) 8 slots, 128 words per slot
(b) 7 slots, 32 words per slot

the PPP model). Thus 20% of the space can be partitioned into pages of 1M word each, while the other 80% can be partitioned into pages of 1K word each. Hence, there can be at most $\frac{2^{32}}{2^{20}} \times 0.2 + \frac{2^{32}}{2^{10}} \times 0.8 = 3356262$ *stacks*, or almost 3.4 million more stacks with ELPS than with static partitioning. While the exact number varies with each program, ELPS can potentially support millions more tasks (each with one or more stacks) than static partitioning without the need for garbage collection or other schemes to handle overflow, while allowing the tasks to fully share the global address space. Currently, we are unable to simulate the full potential of ELPS due to memory limitations imposed by the host on which the simulator is run.

7.2.3 Effect of Page Size on Performance

Page size is a main parameter in the ELPS memory management scheme. It can be set at system configuration time, by specifying the value of the page mask register. In this section, we examine the effect of page size on performance.

Table 7.10 shows the ratio of execution time of ELPS over static partitioning, expressed as an overhead percentage. Seven page configurations are chosen, ranging from 512 words to 8K words per page. From this table, we classified the programs into three groups according to their observed behavior. As the page size increases, the overheads of

programs exhibit three distinct patterns: (1) sharp increase, (2) slight increase, and (3) decrease. (For the time being, consider only the columns marked as "no hashing for data start." The comparison to the "with hashing" columns will soon follow.) Compared to the tasks in the third group, the tasks in the first two groups use fairly small spaces and overflow very infrequently. The overhead increase (as the page size increases) is due to block collisions in the caches. In the third group, the tasks use very large spaces and the overflow very frequently. Thus the larger page size results in fewer overflows and faster execution.

The block collision in the caches can best be explained with the specific cache parameters. Our simulation has a cache configuration of 16K word cache size, 4-way associative, and 4 word block size. Thus, there are 1K cache sets[3] , each set having 4 slots. If the ELPS page size is 1K, the first location of all pages will fall in the same cache set. If the page size is 512, the first location of every other page will fall in the same cache set. In the worst case, a page size of 4K or greater will fall in the same cache set and will occupy all four slots in the set, thus requiring frequent flushes to memory to free up the slots. This flushing increases bus activity and slows down execution. This collision is most serious for programs with very small task spaces (as those in group 1), since most accesses would be done near the stack bases. For programs with larger task spaces (as those in group 2), this collision makes little difference (less than 2%).

There are two possible types of collision in the cache blocks. Internal collision occurs when the low bits of the cache blocks of one stack are identical to those of another stack belong to the same task. This case is most serious for caches where the associativity is less than the number of stacks in a task. External collision occurs when when the stacks belong to different tasks. In our 4-way associative cache, internal collision is not a problem for the four stacks in a task. On the other hand, external collision occurs when a processor executing a new task needs to flush out data from previous task to make room for the new one. When an old task – or a new task which reuses the old space of a terminated task – is swapped in, its stack space needs to be loaded into the cache from memory. Much of this extraneous bus traffic can be reduced by a more intelligent cache system such that: (a) a stack push operation does not need to read the block in from memory, and (b) the cache blocks of a task that has been killed should be invalidated. Such cache system has been implemented on sequential architectures for Prolog (PSI-II [NN87] and Berkeley

[3]number of cache sets = cache size / block size / associativity = 16K / 4 / 4

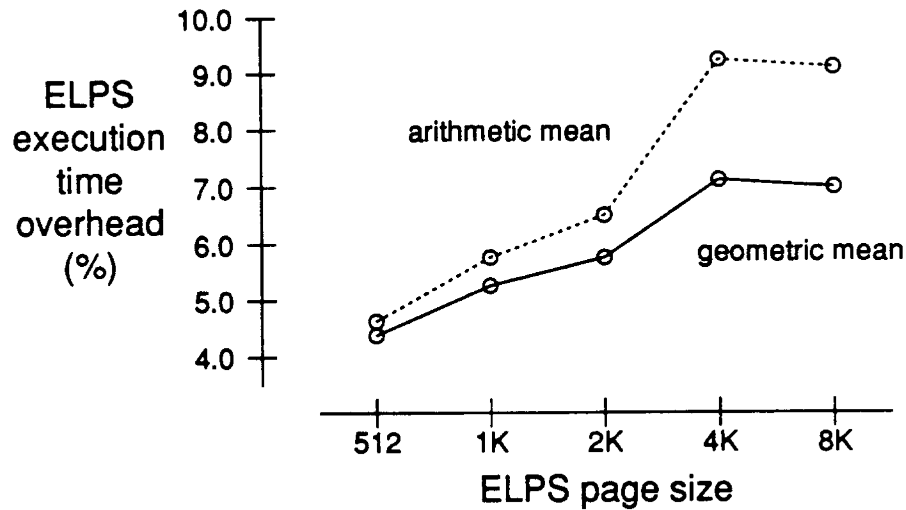


Figure 7.3: Average Effect of ELPS Page Size on Execution Time

Abstract Machine (BAM) [HSC*90]) and on parallel logic programming architectures (by Goto [GMT89]).

Both types of collisions can be controlled by using a hash function to vary the starting address (in an ELPS page) for data storage. The space between the lowest address of the page and starting address for data storage would be left unused. The following hashing function was used to determine the offset for the starting address:

$$offset = (\textit{page number} \bmod \textit{number of slots}) \times \textit{slot size}$$

The two parameters ***number of slots*** and ***slot size*** determines the behavior of the cache collision. A large slot size reduces internal collision, while a large number of slots reduces the external collision. The product (*number of slots* × *slot size*) is the amount of space left unused in each ELPS page.

Consider the two columns labeled as "with hashing" in table 7.10. Compared with no hashing, the hashing scheme eliminates the sharp overhead increase of ELPS in the small programs (qsd200 and query0). Furthermore, various parameters for the hashing function (columns (a) and (b)) make little difference on performance. Thus a small number of slots and a small slot size can be chosen to minimize unused space.

Figure 7.3 graphs the arithmetic mean an the geometric mean of the overheads with respect to page size. Because the page size of 4K is the saturation point (where all four

Table 7.11: Effect of ELPS Page Size on Overflow Frequency

	number of overflows						average overflow handling time (in cycles)					
Benchmark	256	512	1K	2K	4K	8K	256	512	1K	2K	4K	8K
ckt4	0	0	0	0	0	0						
query	0	0	0	0	0	0						
qsd200	10	4	1	0	0	0	22	22	31			
chat	4395	1720	357	0	0	0	2	2	2			
compiler	-	141	26	4	0	0		14	26	39		
tp3	-	375	129	52	21	7		11	15	19	23	40
boyer	-	-	1851	915	458	228			23	26	30	37

Table 7.12: Effect of ELPS Page Size on Internal Fragmentation

	internal (between pages) fragmentation					
Benchmark	256	512	1K	2K	4K	8K
ckt4	0	0	0	0	0	0
query	0	0	0	0	0	0
qsd200	6	2	0	0	0	0
chat	32363	14914	621	0	0	0
compiler	-	776	16	6	0	0
tp3	-	1145	542	81	34	4
boyer	-	-	316	220	104	32

slots in the 4-way associative cache set are filled), the overhead peaks at a page size of 4K. This is also the reason for the sharp overhead increase when the page size increases from 2K to 4K. For boyer and tp3, the overheads continue to decrease as page size increases, while for all other programs, the overheads level off at page size of 4K or greater.

Another important measure of ELPS performance is the frequency of overflows. *Table 7.11* shows the number of overflows and the average overflow handling time for each of the various page configurations. The number of overflows decreases rapidly as the page size increases, as more task spaces can be captured in a page. The average overflow handling time increases slightly as the page size increases, since these overflows are more likely to request a new page. Due to the extremely fast overflow handling time (in tens of cycles), the total time required to handle overflow contributes little to the overall execution time.

Internal fragmentation is the number of words left unused at the end of a page because a structure does not fit on the existing page. *Table 7.12* shows the cumulative internal fragmentation for the different page configurations. Internal fragmentation decreases

very rapidly as the page size increases, and thus is insignificant.

7.2.4 Allocation and Deallocation Strategies

The implemented strategy of on-demand allocation and lazy deallocation performs very well (as expected). For programs with a high degree of backtracking, lazy deallocation is clearly superior to eager deallocation. However, for programs that have a very high space usage, eager deallocation may help distribute the free space more efficiently. In general, the dynamic nature of Prolog tasks makes it very difficult to predict stack usage. A reasonable strategy is to retain one empty page while releasing the others to the free list. This one page buffer would prevent performance degradation due to fluctuations at a page boundary.

7.3 Discussion

With the functionally correct execution of Prolog programs, we have shown the feasibility of ELPS, a hybrid heap-stack mechanism designed to allow efficient sharing of the global address space and efficient space reclamation. The heap style allocation of small segments distributes the limited shared space to where it is needed most, and the stack structure allows for fast space reclamation without the need for garbage collection in many cases. ELPS solves the "sleeper task" problem of PPP, where tasks that have alternative clauses for possible future backtracking tie up the task spaces allocated to them. By allocating only a small amount of space each time, millions more tasks may be created to exploit the potential for parallelism. ELPS has been implemented on a simulated multi for parallel execution of Prolog.

Since all ELPS pages have their starting addresses with the same lower order bits, collision in the cache blocks can potentially be a serious problem. Fortunately, this problem can be solved with a simple hashing scheme to start data storage on a page at various offsets. A more complete solution is to design more intelligent caches for stacks such that: (a) a stack push operation does not need to read the block in from memory, and (b) the cache blocks of a task that has been killed should be invalidated.

The overhead of ELPS is 2% with hardware support and 11% with software only. With a cache system, the overhead of ELPS is around 5% with hardware support, due to some amount of block collisions in the cache. With software only, the overhead remains

at around 11% because the software check is internal to the processor (does not generate cache or bus traffic) and the execution time is dominated by memory access times.

For optimal performance, the ELPS page size should be set at system configuration time such that (a) it is smaller than the number of cache sets to reduce the frequency of collision and cache flushes, and (b) it is sufficiently larger than the largest structure. ELPS provides the memory management needed to keep up with the memory demand for parallel execution, thus increasing the degree of potential parallelism. To obtain a high overall speedup, proper scheduling and granularity control must be coupled with this potential for a very large number of tasks.

In addition to parallel execution of Prolog, ELPS may be used as the memory management scheme for very large scale shared memory multiprocessors which use the multi as a building block. From this viewpoint, ELPS has the following advantages:

- Dynamic allocation and efficient utilization of the shared address space frees the programmer from the concern of memory management. The memory manager can enforce locality by having multiple free-page-lists, one for each group of processors.

- Detection of overflow upon allocation of space on stack provides a more robust system. Some Prolog implementations that use static partitioning has no provision for stack overflow checking.

- The explicit links require no special hardware mechanism for address translation.

- By storing the links with the pages and not together in a table, contention on memory or cache block for different table entries is eliminated.

- By associating a task space with a task and not with a processor, the scheduling of tasks onto processors is more flexible and task migration is more efficient.

For more complete memory management, ELPS may be integrated with a garbage collector to reclaim unused space within each page. In sequential execution of Prolog, most objects do not survive the first iteration of garbage collection [TH88]. In parallel execution, when a task terminates with no more alternative solutions, its entire space can be discarded. Due to the memory usage nature and the highly dynamic life times of the parallel tasks, an local garbage collector should do well in reclaiming unused space and not interfere with the execution of other tasks. One possible approach is to collect only data areas not shared by

other tasks. In the PPP model, this unshared area is the local data space used by a task before it spawns any children.

Chapter 8

Aquarius-II: A Two-Tier Memory Architecture

8.1 Introduction

In the previous chapter, we evaluated the space and time aspects of memory performance of a shared memory bus based multiprocessor. In this chapter, we explore an alternative memory architecture to increase memory bandwidth.

The bus is the simplest interconnection network that implements a "dance hall" shared memory (figure 2.1). With caches local to each processor, this structure is called a "multi" by Bell [Bel85] (figure 5.4). It can provide high performance at relatively low cost. The caches are kept consistent by hardware protocols, thus freeing the programmer from the concerns of managing the memory hierarchy details.

In a multi architecture, two types of memory interference can degrade memory performance:

- *Multiple access interference* occurs when two or more processors need to access main memory. The processors arbitrate for the bus and their memory accesses are serialized. The processors remain idle while waiting for the memory operation to complete.

- *Lock interference* occurs when the bus is being locked up by a processor for an atomic read-modify-write operation. Other accesses to memory are suspended until the bus is released. Such an atomic operation is necessary to synchronize the processors that are executing in parallel.

As the number of processors is increased, the frequency of these memory interferences increases and the bus becomes a more serious bottleneck.

In this chapter, we describe ***Aquarius-II***, a multiprocessor in the "dance hall" shared memory category. Aquarius-II contains two tiers of memory designed to reduce memory interference and to increase the processor-memory bandwidth. The first tier, called *synchronization memory*, is a multi with coherent caches, used to store synchronization data such as locks and semaphores. The second tier, called ***high-bandwidth*** *(HB) memory*, contains a high-bandwidth interconnection network to memory (such as a crossbar). There are HB local caches for each processor, but they are much simpler than the snooping caches of synchronization memory. The HB memory is used to store the bulk of the application program's code and data, as well as the operating system.

8.2 High Performance Memory Architectures

For any architecture, the memory system is potentially a major bottleneck since the access time of a large and economically feasible memory system is 3 to 5 times slower than processor cycle time. This gap is much larger for supercomputers with very short cycle time.

Previous studies [Smi82] have shown that cache memory is a cost effective way to substantially improve performance. For example, the Convex's C-1 [Wal85], a Cray-1S like processor, achieves one fifth the performance of a Cray-1S [SA83] at one tenth of the cost. It uses a large cache (64K bytes), a slower technology (CMOS), a slower memory, and pipelining. The instruction and data caches in the Convex's C-1 play a key role in providing performance even though memory is slow and the memory bandwidth is limited.

Various multiprocessor memory architectures have been employed to obtain high memory bandwidth. Current high-speed multiprocessor systems often contain a fully connected network called the *crossbar*. The hardware cost of a crossbar switch is proportional to the square of the number of processors (assuming an equal number of memory modules). Because of this, the crossbar is used primarily in systems with a small number of processors (e.g., C.mmp and Burroughs B7700 [Sat80]). For a larger number of processors, a multistage network (e.g., the Omega Net in Ultra computer and shuffle exchange in Cedar) is used to reduce cost and to increase fault tolerance. These systems either do not employ caches due to the problem of multiple cache coherency associated with the particular interconnection

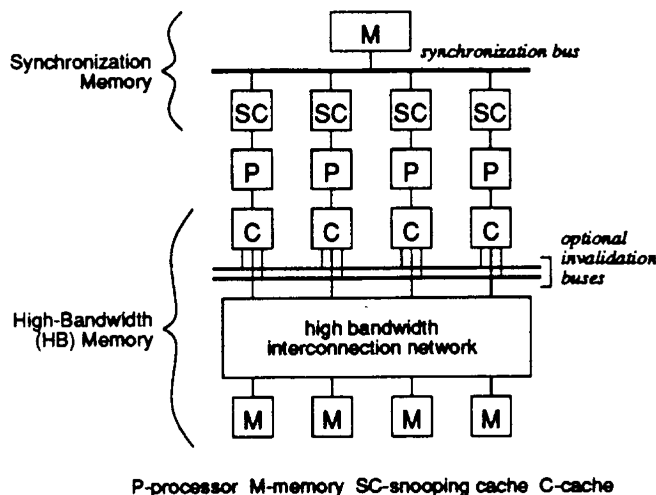


Figure 8.1: The Aquarius-II Multiprocessor Architecture

network, or restrict the use of caches to read-only and non-shared read-write data. The medium-speed multiprocessors, usually called super-minis, contain caches with hardware coherency protocols. To keep the cost low, the caches are connected to a single bus. In this case, the caches are more efficiently utilized, but the single bus connection to memory is a major bottleneck as the number of cache-processor nodes connected to the shared bus is increased.

8.3 The Aquarius-II Architecture

To reduce memory interference and to have both fast synchronization accesses and high-bandwidth data accesses, we propose a two-tier memory system which contains two separate memory spaces: a *synchronization memory* and *high-bandwidth (HB) memory.*

8.3.1 Synchronization memory

The synchronization memory is the upper tier shown in *figure 8.1*. The synchronization caches are connected to synchronization memory via a bus. There have been a number of proposals for multiprocessor cache coherency for a single bus [Goo83, AP84, KEW*85] using a variety of protocols, all of which require monitoring the bus and broadcasting the

data to caches and to memory. Bitar [BD86] has extended Goodman's snooping cache for more efficient locking. Bitar's scheme employs a cache lock state that reduces traffic on the bus, in addition to having one less memory access than the conventional test-and-set scheme for scalars. This scheme requires 3 state bits associated with each cache block and allow for cache-to-cache transfers for update or invalidate. Such a scheme is vital to fast synchronization accesses.

The synchronization memory is used for storing synchronization information and status information. Synchronization information consists of event flags, lock variables, and semaphore variables. The status information contains the status of resources such as processors and buffers, control flags such as modify, reference, and valid used in caches, and mail boxes. The separation of memory results in fast access to synchronizing information since memory requests need not wait for the completion of a long data transfer.

The bus monitoring and broadcasting mechanisms needed to implement fully dynamic cache coherency protocols requires complex hardware, and the single shared bus to synchronization memory can be a serious bottleneck. The smaller sizes of the synchronization memory require simpler hardware circuitry (for cache and memory designs) that result in lower design cost and faster address decoding. Since synchronization memory is used only for synchronization and status information, it should be a small fraction of the total memory address space (less than 10%).

8.3.2 High-bandwidth Memory

The vast majority of the memory space (90%) is in the second of the two tier memory, called the ***high-bandwidth memory***. This second tier provides a very high processor-memory bandwidth by using a high-bandwidth interconnection network to connect the processors to the memory modules. For a multiprocessor system with a small number (16 or less) of processors, a crossbar is most appropriate since it provides the highest possible bandwidth with reasonable cost. The crossbar switch should contain an arbitration unit that resolves conflicting memory requests to the same memory bank in a fair manner (starvation free) [Sri88]. For systems with higher number of processors, a lower performance but less expensive interconnection network such as the Omega network [Gea83] is more suitable.

To further enhance the performance of the high-bandwidth memory, a cache is placed in between the processor and the interconnection network. These caches greatly

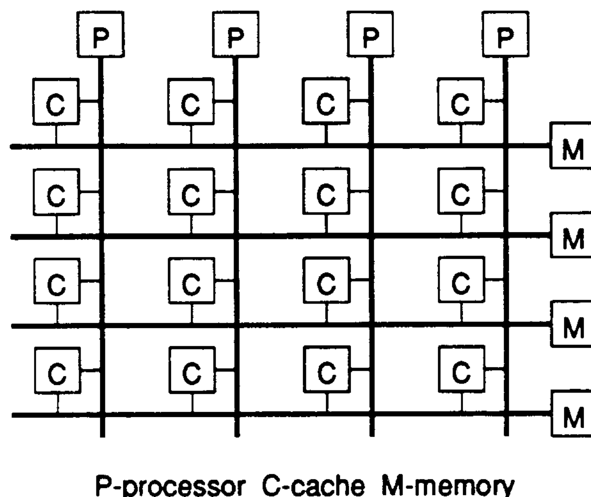


Figure 8.2: Multiprocessor Architecture with Caches at Each Crossbar Switchpoint

increases the complexity of the memory system due to the cache coherency problem. In order to keep the crossbar switch as simple as possible, we have chosen to put a cache at each processor (*figure 8.1*) instead of a cache at each switch point of the crossbar, as shown in *figure 8.2*. The topology of figure 8.2 is equivalent to having a very high number of buses, each of which having the complexity of a synchronization bus.

8.3.3 High-Bandwidth Memory Cache Coherency

A number of solutions have been considered for the cache coherency problem for the high-bandwidth memory. The goal is to avoid the hardware cost and complexity of the full snooping and cache-to-cache transfer protocols. In this section, two such schemes are discussed: software scheme by restricted caching and hardware broadcast for invalidation.

Restricted Caching

The simplest way to resolve the cache coherency problem is to avoid it completely by restricting caching to read-only and non-shared read-write data. Software arrangement of data space, combined with some hardware support in the individual caches, is done to make sure copies of a writable cache block is not allowed in more than one cache at any

given time.

This restricted caching scheme, called *singly cacheable protocol*, can be implemented with each cache having a *cacheable address table* (using associative memory) which contains the address ranges that are cacheable by that cache. If a memory block is *cacheable* by a particular cache, when a cache miss occurs, that data block is fetched from memory and is stored in the cache. Subsequent reads to this block gets data directly from the cache, and subsequent writes may either write-back to cache only (if the data is local to a task), or write-through to both the cache and the memory (if it is potentially used by other tasks). If a memory block is *non-cacheable* by a particular cache, it is never stored in the cache and all references to it require going to the memory.

The cacheable address table in each cache contain the address ranges for the code space (read-only) and the address ranges of the local data spaces of the tasks that are assigned to the processor. The scheduler assigns a task to a processor, and the task is to be executed by that same processor until termination. When a task is moved from one processor to another (task migration), the cache in the old processor must be flushed, and the task's local data space address ranges must be moved from the old processor's cacheable address table into the new processor's cacheable address table.

The main advantage of this restrictive caching scheme is that no communication among the caches is necessary, avoiding the need for complex circuitry for bus monitoring and broadcasting. Furthermore, the caches are completely independent of each other, and do not interrupt each other for invalidation or update. For some applications, these advantages may be overshadowed by the performance penalty of higher miss ratio due to non-cacheable blocks and more accesses to memory are necessary for the write-throughs. However, the crossbar provides some relief by having a very high processor-to-memory bandwidth to handle this memory traffic.

Broadcast for Invalidation

To increase the cache hit rate, the HB caches can be extended to allow caching of all memory blocks. Instead of the full coherency mechanisms used for the synchronization caches, simpler invalidation buses can be used to keep the HB caches consistent. Upon a cache read miss, the data block is loaded into the cache from memory. When this data block is modified, the cache broadcasts this write by putting the address of the modified

block onto the invalidation bus. The other caches constantly monitor this invalidation bus for an address in their own directories. If a cache contains a copy of the data block, it will invalidate its own copy, and its next access to this data block has to get data from memory. If the cache does not contain a copy of the data block, its normal operation is uninterrupted by the traffic on the invalidation bus. Furthermore, invalidation involves only clearing the valid bit for the block in the cache directory, and may be done concurrently with some other cache accesses which do not alter the directory (assuming 2 read ports for the cache directory). Bus traffic can be reduced by having a *private* cache state for cache blocks owned by only one processor. No broadcast for invalidation is needed for writes to private cache blocks.

The separation of the broadcast mechanism from the crossbar keep the crossbar switches simple while allowing a flexible number of invalidation buses to be used. Each invalidation bus is much simpler than the synchronization bus since it is only for the address, whereas the synchronization bus is for both address and data. Each invalidation bus is used for a different address range, so a simple demultiplexer can be used to choose the appropriate bus for the address to be broadcasted. If there are multiple requests by the caches to broadcast on the same invalidation bus, these caches must arbitrate for the bus in the same way that the synchronization caches arbitrate for the synchronization bus. With a greater number of in validation buses, each bus will cover a smaller address range, thus resulting in less probable contention for any given bus. Each invalidation bus requires a bus monitor unit in each of the caches to monitor traffic on the invalidation line.

8.4 Parallel Execution of Prolog

Studies by Eggers and Katz [EK87] analyzed the memory reference patterns of write shared data in several parallel applications coded in FORTRAN. Their trace simulation results show very small percentages of write shared data (less than 2%). For the two-tier memory system, this would mean very low demand on synchronization memory and thus the one synchronization bus is sufficient to support this traffic without much contention. We wish to analyze a more complex programming paradigm to evaluate how it can be supported architecturally with the two-tier memory model. Our language of choice is Prolog. In this section, we present the usage for the two-tier memory system for parallel execution of Prolog. In particular, the PPP execution model [Fag87] is analyzed for

implementation on the Aquarius-II.

8.4.1 Synchronization Characteristics

Prolog has some unique features compared to the languages of other programming paradigms such as Lisp (functional programming) or FORTRAN (imperative programming). Prolog variables are single assignment, which once given a value will never be changed. This is very suitable for parallel processing since a variable that has been written becomes read-only, and no synchronization for writing is required. This can be characterized as *single writer, multiple readers*, and *read-only after write*.

Except for the global database in Prolog which resides in the code space, all Prolog variables are local to a Prolog clause and are explicitly passed from one procedure to another. This simplifies the task of detecting the communication between parallel tasks in which the procedures are executed. For the rest of this section, we review the PPP execution model that was previously described in section 5.4.4 (Task Kernel Module).

The PPP execution model contains two kinds of tasks: AND tasks and OR tasks. Each task has a task control block, called a *task table entry*, which contains the state of the task, base addresses of the task data space, and various links to the parent task. Each task also has its own data space for private and read-shared data.

OR-tasks are used to execute the multiple clauses of a procedure. Each OR-task contains its own binding area (called a *hash window*) to store the variable bindings that would conflict with other OR tasks. These bindings are seen by the task's parent by means of dynamic linking of the hash windows. Even though the clauses of a procedure are executed in parallel and out of order, the results obtained from them are serialized to maintain standard Prolog semantics.

AND tasks are used for executing AND-parallel subgoals that are independent of each other; that is, they cannot attempt to bind the same variable during execution. Subgoal independence is determined statically, by programmer's annotations and/or by data flow analysis (such as *static data dependency analysis (SDDA)* [Cha85, Cit88]; the general technique is known as *abstract interpretation* [BJCD87, WHD88]). The AND tasks do not have their own hash windows, but instead they share the hash window of their closest OR ancestor.

The task creation, context switching, and termination tasks are distributed among

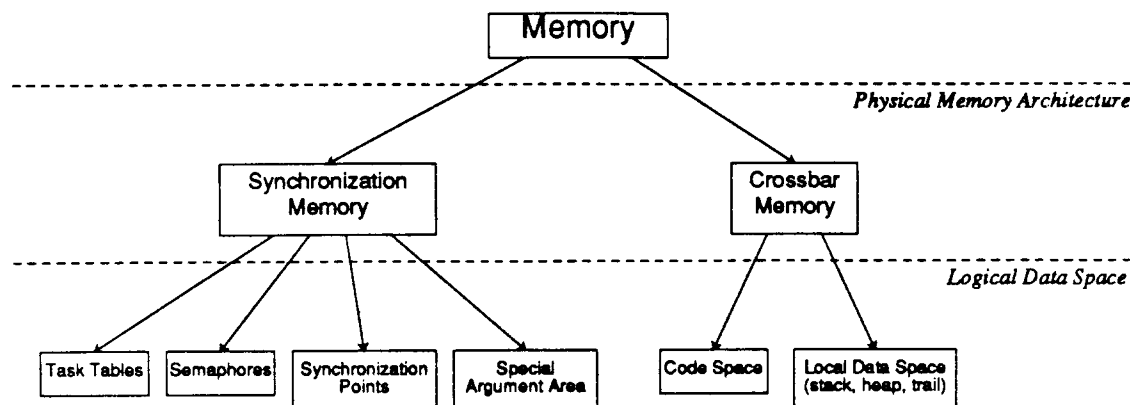


Figure 8.3: Mapping of the PPP Storage Model onto the Two-Tier Memory System

the processors, synchronized by mutually exclusive accesses to shared queues. A processor executing a task performs all the tasks needed to spawn a child task, from reserving task space in memory to writing out the new task state and putting it in a ready-queue. An idle processor removes a runnable task from a ready-queue for execution. A terminating task frees up its own task space by putting the task space in a free-task-list. A task goes to sleep by swapping itself out of the processor, and gets awakened by another task moving it into the ready-task-queue. The tasks communicate via synchronized accesses to shared memory.

8.4.2 Mapping of PPP onto Aquarius-II

Figure 8.3 shows the mapping of the code and data spaces of the PPP Execution model onto the two-tier memory system of Aquarius-II. The task tables, the hash windows and the global heap (containing join tables and children task identifiers) are stored in the synchronization memory, either because these data are shared writable and require locking or because they are modified often and need the cache to cache data transfer capability of the synchronization bus.

On the other hand, the code space and the local data space reside in the high-bandwidth crossbar memory. The code space is for the most part read-only, and the local data space is mostly locally writable by the owner task, and read shared by other tasks.

8.5 Chapter Summary

In this chapter, we described the two-tier memory architecture of the Aquarius-II, designed for both fast accesses to synchronization data and high memory bandwidth. The first tier, called *synchronization memory*, is a multi memory architecture, with caches local to each processor and connected to memory via a single bus. The caches are kept consistent using hardware coherency protocols. The second tier, called *high bandwidth (HB) memory*, contains caches local to each processor connected to memory via a crossbar. Two general coherency protocols have been proposed to keep the crossbar caches consistent: restricted caching and broadcast for invalidation. Restricted caching is done primarily in software, while the broadcast scheme needs an additional invalidation bus and bus monitoring circuit.

Chapter 9

Aquarius-II Simulation Results

How much performance improvement can be obtained by separating the synchronization data from others? This can be answered by evaluating the bus traffic and the synchronization behavior of parallel execution. This chapter provides the simulation results of the two tier memory in comparison with the single bus memory.

9.1 Simulation Parameters

To compare the performance of a two-tier memory architecture to a single bus cache coherent system, the NuSim simulator (described in chapter 5) is set up to simulate two such systems. The multiprocessors are configured to 8 processors. The cache of the *Single Bus (SB)* memory is 64 Kbyte, 4-way associative, with 16 byte blocks. The relative speed of the bus is set at a (fast) 1 processor cycle for arbitration and two cycles for broadcasting. Main memory is also set to be very fast, requiring 2 cycles for first byte and 1 cycle for subsequent bytes in a block.

The *Two-Tier* system consists of the synchronization bus (SB) memory, which is identical to the Single Bus system above, plus the high bandwidth (HB) memory containing the crossbar. A memory request from the processor is multiplexed into one of the two tiers of memory (simultaneous access to both tiers is not allowed). Lock/unlock accesses and read/write accesses to shared data structures are channeled into the SB memory, while all other memory accesses are channeled into the HB memory. Two simplifying assumptions are made regarding the crossbar: no memory bank conflict and single cycle access delay. These assumptions are made to study the maximum improvement potential of the two-tier

Table 9.1: Access Ratios for Shared and Local Memory Areas

Benchmark	Percentage over all types of accesses							
	code		local data		shared data		locks	
	count	time	count	time	count	time	count	time
boyer	42	22	42	33	13	37	3	9
chat	46	50	54	50	0*	0*	0*	0*
ckt4	31	16	44	38	20	34	5	12
compiler	28	33	43	42	24	21	4	4
qsd200	45	21	43	34	9	36	2	10
queens6	42	22	48	35	5	18	1	5
query	45	27	28	16	12	22	1	4
tp3	51	41	42	41	6	15	1	3
average	41	29	43	36	11	23	2	6

* close to 0, much less than 1

Table 9.2: Average Access Time for Shared and Local Memory Areas

Benchmark	Average access time (in cycles)			
	code	local data	shared data	locks
boyer	1.7	2.5	9.3	10.9
chat	1.2	1.1	7.5	6.9
ckt4	1.0	1.8	3.4	5.0
compiler	1.5	1.2	1.1	1.1
qsd200	1.2	2.0	9.6	12.9
queens6	1.1	1.5	7.1	12.3
query	1.5	1.4	4.5	9.8
tp3	1.2	1.4	3.9	4.5
average	1.3	1.6	5.8	7.9

architecture over the single bus architecture.

Eight Prolog benchmark programs are chosen for our simulation study. These programs exhibit a variety of parallelism characteristics and locking behavior. *Table 7.7* (in chapter 7) showed the list of benchmarks, their static code sizes, and a brief explanation for each.

9.2 Memory Access Behavior

To observe the memory behavior of parallel execution of Prolog, the benchmark programs were run on the single bus system. *Table 9.1* shows the access ratios for the various

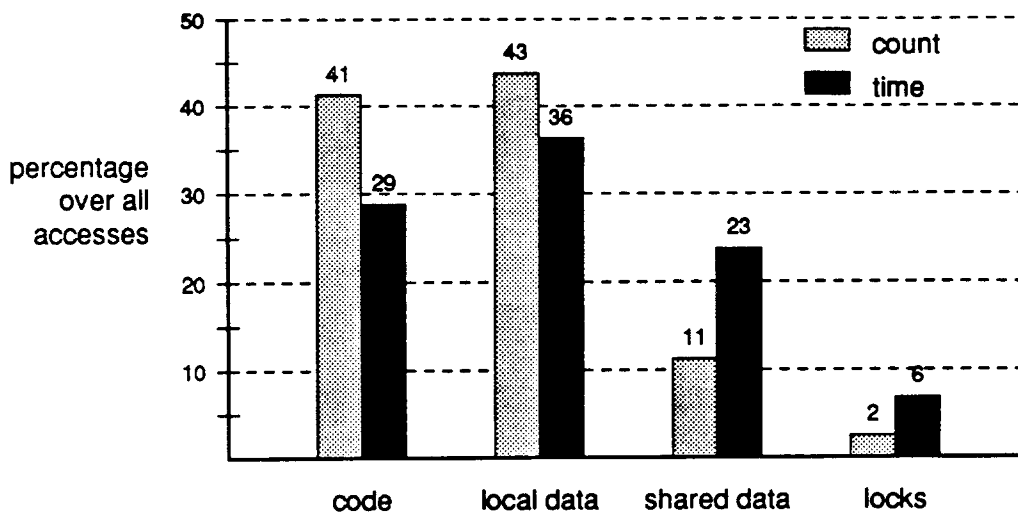


Figure 9.1: Average Access Percentages of Shared and Local Memory Areas

shared and local memory areas while ***table 9.2*** shows the average access time of accesses to each memory area. *Count* is the number of accesses and *time* is the access time required. The values in table 9.1 represent the percentage of accesses for that area over all areas. For example, 42% of the memory accesses in boyer are to the code area, but they require only 22% of the total memory access time. On the average, code access is frequent and fast (41% of count, 29% of time, and 1.3 cycles per access). This is because code is read only and may be reused often (in the case of recursive calls), resulting in better cache performance. Compared to code access, local data access is a little more frequent (43% count) and a little slower (1.6 cycles) while shared data access is much less frequent (11% count) and much slower (5.8 cycles). Lock access is the least frequent (2% count) and the slowest (7.9 cycles). The access count and the access time for locks depend on the synchronization needs of the programs. Ckt4 requires frequent communication among parallel tasks that explore different circuit designs, while the chat parser has tasks that operate on independent sentences, requiring little communication. Compiler shows an interesting behavior in that the average access time for shared data and locks is only 1.1 cycles. This is because the shared data is the very large block of object code generated by the compiler. Access to this block is very localized, with each task contributing a small portion to make up the whole block. Block copying is very fast and has high cache hit ratios.

Table 9.3: Execution Time and Bus Utilization of Single Bus vs. Two Tier

	cycles		speedup	bus utilization		
Benchmark	single bus	two tier	$\frac{1\ bus}{2\ tier}$	single bus	two tier	% change
boyer	74016	58002	1.28	0.75	0.58	-23
chat	2353059	2271477	1.04	0.09	0.00	-100
ckt4	1866705	1656694	1.13	0.40	0.29	-27
compiler	1261500	1092355	1.15	0.19	0.03	-84
qsd200	97048	80502	1.21	0.50	0.37	-26
queens6	218178	188748	1.16	0.72	0.66	-8
query	63586	55417	1.15	0.76	0.64	-16
tp3	295042	257045	1.15	0.16	0.08	-50
average			1.16	0.45	0.33	-26

The bar graphs in *figure 9.1* show the relative frequency of accesses among the four memory areas and the relative percentages of time required. For code and local data, the time bar is below the count bar. For shared data, the time bar is twice that of the count bar. For locks, this ratio is three. While locks make up from much less than 1% to 5% of the access count, they can take up to 12% of the access time. More importantly, they can hold up the bus, thus blocking out other unrelated accesses (particularly those accesses to local data). The two tier memory resolves this problem by diverting local data accesses to the HB memory.

9.3 Execution Time of Single Bus vs. Two Tier

Table 9.3 compares the execution time and the bus utilization of the single bus with those of the two tier memory. The speedup column shows the ratios of single bus execution time over two tier execution time. With code and local data accesses diverted to a different path, execution on the two tier memory shows modest speedups of 1.04X to 1.28X (1.16X on the average). As expected, those programs with the highest percentages of lock accesses show the greatest speedups. The (bus utilization) *single bus* column shows the potential bottleneck of the single bus memory, while the *two tier* column shows the bus utilization due to shared data and locks only. The last column shows the percentage of change (decrease) in bus utilization when the two tier memory is employed. The percentage of bus accesses due to code and local data range from only 8% for queens6 to nearly 100% for chat. The average decrease in bus utilization is 26%. This explains the long access times

for shared data and locks compared to code and local data. Programs that have a high percentage of bus utilization due to code and shared data benefit the most with the two tier memory.

9.4 Parallel Execution Behavior

Figures 9.2 and *9.3* show the run times of the tasks generated in the execution of two programs: quicksort and queens6. In each graph, the vertical coordinate contains the task IDs, ranging from 0 to 63, and the horizontal coordinate contains the time in units of processor cycles. Each continuous horizontal segment represents a period of time in which the task is being run in a processor. There are at most 8 tasks executed at any given time, one in each processor. The other tasks wait in the ready queue for idle processors.

The graphs of the two benchmarks are selected to show varying behavior of parallel execution: quicksort running in AND-parallel and queens6 running in OR-parallel. As shown in figure 9.2, quicksort behaves very regularly, with each task spawning one additional task to work on one of two partitions, while the parent task continues execution with the other partition. The join operation occurs after each of the subpartitions has been sorted. The long segment for task 0 starting at time 0 indicates that a large chunk of time is spent in sequential execution, and that is a deterrent to overall speed up when the number of processors is increased. With 8 processors, parallel execution on single bus results in a 2.1X speed up over sequential execution. Parallel execution on the two-tier results in 2.6X speed up, approaching the theoretical limit of 3X speed up for a balanced tree ($log_2 8$, for 8 processors).

On the other hand, queens6 executes in a quite random manner, as shown in figure 9.3. The program searches for board positions to place non-attacking queens. It backtracks to alternative search paths when the current path fails to give a solution. Space and IDs of terminated tasks are reused for new tasks. There is a high degree of parallelism in queens-6, but the overall speed up is limited by the number of processors, the contention on the single bus and the adherence to standard Prolog semantics (ordering the returned solutions of the OR-tasks from left to right). With 8 processors, parallel execution on single bus is 4X faster than sequential execution, while parallel execution on two-tier is 4.6X faster than sequential execution.

For each benchmark, the patterns of parallel execution on single bus and on two

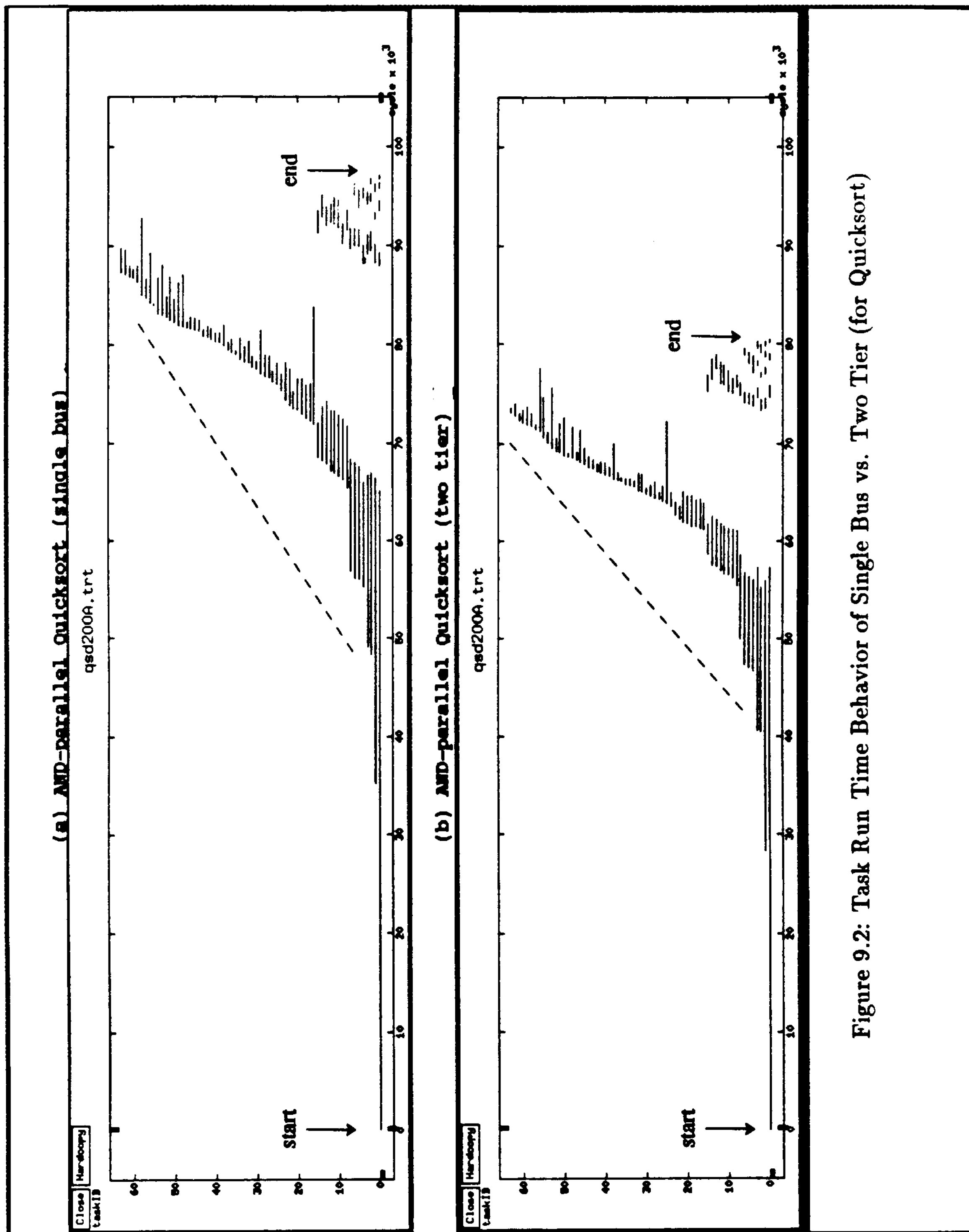


Figure 9.2: Task Run Time Behavior of Single Bus vs. Two Tier (for Quicksort)

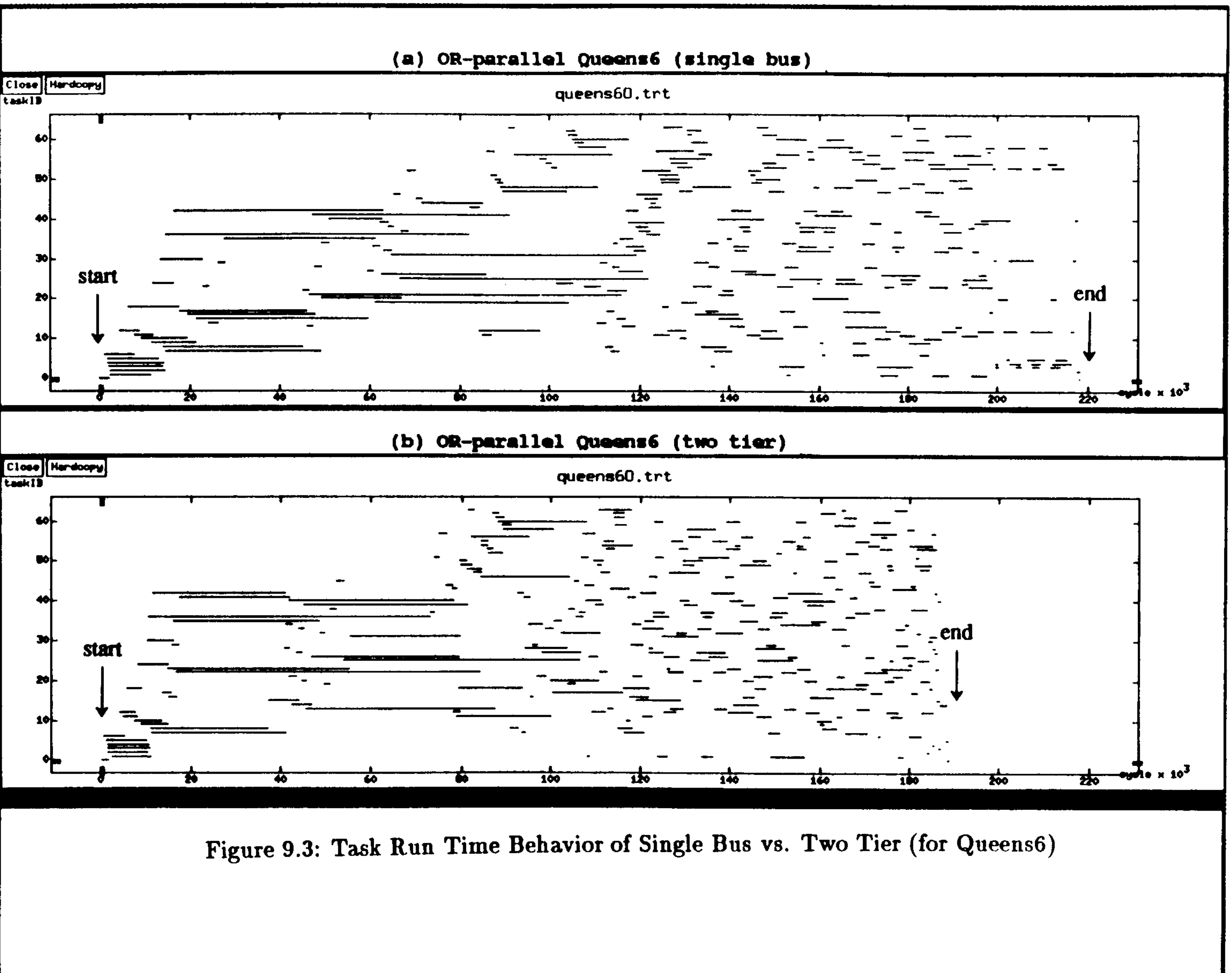


Figure 9.3: Task Run Time Behavior of Single Bus vs. Two Tier (for Queens6)

Table 9.4: High Bandwidth Memory Access Locality

	Percentage over all accesses				
	internal		external		est. max
Benchmark	read	write	read	write	hit ratio
boyer	69.3	20.9	7.6	2.2	90.2
chat	68.1	23.2	6.2	2.5	91.3
ckt4	69.6	15.0	9.8	5.6	84.6
compiler	66.5	22.1	8.9	2.5	88.6
qsd200	62.8	16.9	9.9	10.4	79.7
queens6	74.4	19.7	4.7	1.2	94.1
query	81.5	7.0	9.1	2.4	88.5
tp3	61.5	12.2	15.5	10.8	73.7
average	69.2	17.1	9.0	4.7	86.3

tier memory are very similar. However, they differ in the starting times and lengths of execution. The dashed diagonal lines in figure 9.2 show the slopes of the starting execution times of new tasks. Note that the slope of the line in figure (b) (two tier) is slightly steeper than that of figure (a) (single bus). By reducing memory contention due to locks and shared data, the two-tier memory allows faster task creation and shorter execution times, resulting in overall faster execution.

9.5 Crossbar Cache Performance

In section 8.3.3, we discussed two schemes for keeping the caches of the high-bandwidth memory consistent. In this section, we evaluate the performance of these schemes.

9.5.1 Restricted Caching

Each AND-task or OR-task has a local task space. A memory access is *internal* if the accessed address is inside this local task space; otherwise, the memory access is *external*. Under the restricted caching scheme, only code and internal accesses are cached. Other accesses result in cache misses and must obtain data from memory, via the crossbar. Thus, a higher percentage of internal accesses would result in a better cache hit ratio.

Table 9.4 shows a breakdown of the accesses in four categories: internal read, internal write, external read, and external write. The internal read column contains code

and internal local data accesses. The percentage of internal read varies from 61.5% to 81.5% and the percentage of internal write varies from 7.0% to 23.2%. This depends on the nature of the benchmark. For example in quicksort, both external read and external write make up high access percentages. This is because the two partitioned sublists to be sorted in parallel are stored in the parent's local space when passed to the children, thus reads to the these original sublists are external. After the sublists are sorted, the parent task does an external write when it links the two sorted sublists stored in the children's local space. Queens6 shows the other extreme behavior: the tasks that explore the board positions mostly read and write in its own local space.

On the average, internal read makes up the vast majority (69.2%) of all accesses, followed by internal write at 17.1%. Both types of internal accesses together make up an average of 86.3% (shown in the last column). This number is also the maximum hit ratio obtainable for the crossbar caches. The actual hit ratio will be somewhat lower, depending on the cache parameters such as cache size, block size, and associativity. The degradation in performance due to a low hit ratio of restricted caching is offset by the high bandwidth of crossbar. Consider the following back-of-the-envelope calculations to compare performance of single bus versus crossbar. The approximate bus bandwidth can be computed as follows:

$$bus\ bandwidth = \frac{width\ of\ data\ transfer\ (in\ bytes)}{data\ transfer\ time\ (in\ nanoseconds)}$$

With the same width of data transfer and data transfer time for the crossbar, the maximum crossbar bandwidth is P times greater than bus bandwidth, where P is the number of processors, assuming that there are more memory modules than processors and that there are no bank conflict. Thus, the miss ratio of the crossbar cache can be up to P times higher than that of the single bus cache to provide the same performance. In our simulation example, P is 8, the average cache miss ratio is 3.9% for full caching and 13.7% for restricted caching. Thus maximum potential of 8X increase in bandwidth by using a crossbar is sufficient to offset the 3.5X increase in miss ratio.

9.5.2 Broadcast for Invalidation

In section 8.3.3, broadcast for invalidation was proposed as an alternative to restricted caching. This scheme requires an invalidation bus connecting the caches together and a bus monitor built into each cache (a simple bus snoop mechanism). When the processor writes to a cache block which it does not have exclusive ownership, the address of

Table 9.5: Write Percentage of Local Data

Benchmark	number of reads & writes	number of writes	write ratio (%)
boyer	48223	22340	46.3
chat	517325	250403	48.4
ckt4	193986	65504	33.8
compiler	571617	245296	42.9
qsd200	54997	30816	56.0
queens6	301510	118270	39.2
query	40346	9940	24.6
tp3	70818	36024	50.9
average			42.8

the block is broadcasted on the invalidation bus and copies of the block in other caches are invalidated. In this section, we evaluate the performance of this broadcast scheme.

Trace simulation is used to evaluate the crossbar cache performance. The cache trace simulator used is a multiprocessor extension of *Dinero III*, a uniprocessor cache simulator developed by Hill [Hil87]. Traces of local data accesses are collected during parallel execution on NuSim, in a four-tuple of (read/write, address, processor ID, timestamp). The traces are sorted in the order of the time stamps and fed into the multiple caches.

In general, trace simulation has limited usefulness for multiprocessor cache evaluation [Bit89b]. In our case, trace simulation provides a good approximation of true performance due to the following conditions: (1) local data traces have little sharing among them, and (2) no synchronization is needed for local data accesses.

Table 9.5 shows the number of cache accesses and the percentage of cache writes. For *boyer* and *tp3*, only the period of active parallel execution is captured. During the other times, the execution is practically sequential and is uninteresting for multiprocessor cache study. Due to limited disk space to store the traces and limited processing power available for simulation, only the first half million accesses are captured for *compiler* and *chat*. A few simulation runs with larger traces for these two programs indicate that the half million traces are sufficiently indicative of the worst case performance. Other programs are run to completion. On the average, writes make up 42.8% of all accesses to local data (reads make up the other 57.2%). Compared to other languages, this fairly high percentage of writes is due to the nature of Prolog and the use of structure copying by the WAM model.

Table 9.6 shows the performance of the broadcast for invalidation scheme. The

Table 9.6: Performance of Broadcast for Invalidation Coherency Scheme

	restricted caching	broadcast for invalidation			
Benchmark	hit ratio (%)	hit ratio (%)	number of writes	number of broadcasts	broadcast ratio (%)
boyer	90.2	95.6	22345	320	1.4
chat	91.3	97.5	258834	9	0.0
ckt4	84.6	97.5	65979	6443	9.8
compiler	88.6	97.7	217879	146	0.1
qsd200	79.7	96.9	31465	106	0.3
queens6	94.1	98.9	121207	1785	1.5
query	88.5	97.8	9993	45	0.5
tp3	73.7	97.3	36244	131	0.4
average	86.3	97.4			1.8

restricted caching hit ratio column is duplicated from table 9.4. The next four columns present performance measures for the broadcast scheme. Overall, broadcast for invalidation yields a much higher hit ratio than restricted caching. More importantly, it supports data sharing among tasks more efficiently by being relatively insensitive to internal versus external access. The hit ratio column of the broadcast scheme contains the best obtainable hit ratios for direct mapped caches with sizes ranging from 4K to 256K bytes. For most programs, the hit ratio peaks at 32K or 64K; for some, the hit ratio continues to increase slightly beyond 32K. The average hit ratio of a 2-way associative cache is 97.6%, which is 0.2 percentage point higher than that of a direct mapped cache of the same size. Overall for caches of size 32K or larger, 4-way associativity yields no better hit ratio than 2-way associativity.

The broadcast ratio provides a measure for the degree of read sharing and task migration (recall that when a processor writes to a cache, that cache broadcasts an invalidate signal on the bus if it does not have exclusive ownership, i.e., one or more other caches contain copies of the block). The small broadcast ratios (most less than 2%) indicate that there is little sharing. One exception is ckt4, which has a broadcast ratio of 9.8%. In this case, ckt4 contains many tasks that terminate quickly, and the old task spaces are reused for the newly created tasks. Since the scheduling is quite random, a new task may get picked up by a different processor (similar to task migration), and the task space in the old processor's cache is invalidated. The extremely small broadcast ratios indicate that one invalidation bus should be sufficient to handle the broadcast traffic.

9.6 Discussion

In the previous chapter, we described the Aquarius-II multiprocessor system with the two-tier memory architecture, designed to reduce lock contention and provide very high processor to memory bandwidth. In this chapter, we have presented the simulation results of parallel execution of Prolog on the Aquarius-II.

Although accesses to write-shared data and locks make up only 13% of the total number of accesses, they account for 29% of the total access time. With other coherency protocols which provide less efficient locking than the cache lock state protocol, the access time is even worse. By separating accesses to write-shared data and locks from accesses to code and other read-shared data, bus contention and bus traffic can greatly be reduced. Access time can be further improved by providing high bandwidth to memory using a crossbar. The two-tier memory architecture of the Aquarius-II provides an average speedup of 1.16X; as expected, programs with a high degree of synchronization benefit most from the two-tier memory. In addition to the degree of parallelism that exists in the algorithm, the speedup depends heavily on the scheduler to maximize the hit ratios of the crossbar caches, and on the memory manager to minimize crossbar memory bank conflicts.

With respect to coherency for the crossbar caches, restricted caching results in relatively low hit ratio (86.3% on average). Fortunately, the high bandwidth of the crossbar makes it comparable with unrestricted, full snooping schemes on single bus. Unfortunately, it is extremely sensitive to the degree of read sharing among the tasks and task migration. Broadcast for invalidation provides a more complete solution to the coherency problem. The measured average hit ratio of 97.4% is much better than that of restricted caching, and the scheme is more suitable for task migration and data sharing. If the scheduler takes into account the previous processor that executes a task, it can reduce task migration and thus greatly increase the hit ratio.

The Aquarius-II may also be used for programming paradigms other than logic programming. Its shared, high bandwidth memory architecture should make it suitable for memory intensive applications that also require a high degree of synchronization accesses during parallel execution. The Aquarius-II may be used for a hybrid imperative/logic programming paradigm, such as a C program invoking Prolog routines for symbolic computation.

Chapter 10

Conclusion

10.1 Summary and Contributions

In this dissertation, the main focus has been on space distribution for a vast number of parallel tasks executing in a shared memory multiprocessor. A new shared memory multiprocessor has also been proposed to increase memory bandwidth and to reduce bus contention due to synchronization. The contributions of this dissertation are as follows:

- *A hybrid heap-stack scheme, called* **ELPS** *(Explicitly Linked Paging Stack), was proposed for managing a globally shared space for parallel execution of Prolog.*
  The dynamic allocation strategy of ELPS supports efficient sharing of global space, thus allowing a very large number of tasks to be created for exploiting the full parallelism potential (as described in section 7.2.2). The obtainable speedup is dependent on the scheduler of the execution model. With hardware support, ELPS incurs an execution time overhead of 2% for a single cycle memory system. When caches are taken into consideration, ELPS overhead increases to 5% due to collision of the blocks in the cache. A hashing scheme was used to greatly reduce the collision. With software only (no hardware support), the overhead of ELPS is less than 11% on average, including the effects of cache collisions.

- *A shared memory multiprocessor, called* **Aquarius II**, *was proposed for fast synchronization and high memory bandwidth.*
  The memory architecture contains two tiers: the upper tier (called synchronization memory) has local caches connected to memory via a bus and the lower tier (called

high bandwidth memory) has local caches connected to memory via a crossbar. This architecture provides an average speed up of 1.16X by reducing contention on the bus and by providing high bandwidth to memory. Programs with a high degree of synchronization benefit most from such an architecture. Two coherency protocols for crossbar caches were evaluated. Compared with restricted caching, the broadcast for invalidation scheme provides much better hit ratios (97.4% versus 86.3% on average). More importantly, the broadcast scheme provides more efficient support for data sharing and task migration.

- *A flexible event-driven simulator, called* NuSim, *was developed to simulate the multiprocessor system at various levels.*
  The modules in the simulator represent the parallel execution model, the processor (and its microcode), and the memory system. The memory system includes the cache coherency protocol. The features of NuSim include multi-level debugging and the capability to execute large benchmarks. This simulator was instrumental in evaluating the performance of ELPS and Aquarius-II.

10.2 Future Work

This dissertation has provided valuable insights into the tradeoffs of a dynamic allocation scheme as an alternative to other approaches. Complete memory management for parallel execution should include a combination of approaches. Future work in memory management can be extended to include the following:

- *reduction of ELPS overhead.*
  With the simple hashing scheme described in section 7.2.3, collisions in the cache blocks are less frequent, resulting in an average of 5% overhead. With a more careful mapping of the pages, this overhead can be reduced to 2% (when no caches are used) or even less with improved scheduling. This overhead reduction is especially important for newer execution models of Prolog, in which the sequential engine is more efficient than the WAM.

- *garbage collection.*
  A local garbage collection scheme which garbage collects only sections of data known to be unshared should do well in reclaiming unused space and should not interfere

with other busy processors. A new ELPS page can be quickly allocated should the current space overflows even after garbage collection.

- *variable ELPS page size.*

  For the chosen benchmark set, the overflow frequency and overflow handling time are insignificant. Should this become a problem, the ELPS page size could be doubled for each time a stack overflows. However, the advantage of reduced overflow frequency may not be sufficient to overcome the disadvantages of the extra overhead for bookkeeping variable sized pages and the space left unused in a large page.

- *multiple free page lists.*

  As the overflow frequency increases, the single free page list becomes a bottleneck. Multiple lists may be kept to reduce this bottleneck and to increase the locality of the pages with respect to a processor.

- *improved scheduling.*

  By taking into account the processor that previously executes a given task, the scheduler can keep a task local to a processor as much as possible to increase better cache performance. Furthermore, if the scheduler can provide an approximation on the size and nature of space usage by a task, more appropriate memory management measures can be taken (e.g., variable page size, incremental garbage collection).

- *virtual memory.*

  In this dissertation, ELPS is considered to be implemented on a system with no virtual memory. When ELPS is implemented on top of virtual memory, the factors to be considered include: ELPS page size, bookkeeping strategy and allocation strategy. The ELPS page size should be a multiple of a virtual page (and hence a multiple of a physical page frame), since much of the space in a page may be left unused. The ELPS free page list maintenance and page allocation should be done to minimize the number of page faults (this implies that pages should be reused as much as possible).

The work on the Aquarius II presented in this dissertation is only a preliminary evaluation of this two tier memory architecture. The success of this architecture depends heavily on ability to build a fast and inexpensive crossbar which is competitive with state of the art buses. Future work on this two-tier memory architecture may include:

- *a detailed study of bus designs.*

  There various techniques that can increase effective bus bandwidth (e.g., pipeline requests) and speed up response time of a cache miss (e.g., cache bypass). These features may increase the bus performance to the level of a slow crossbar.

- *a detailed study of crossbar designs.*

  The low-latency crossbar chip designed by Srini [Sri88] is a good candidate for detailed simulation studies. Advanced VLSI technology allows such complex circuitry to be mass-produced at low cost.

- *mappings of task spaces onto crossbar memory modules.*

  To achieve the highest potential bandwidth of a crossbar, the memory spaces used by the parallel tasks should be mapped onto the memory modules of the crossbar in a way such that module conflicts are minimized.

- *other parallel programming paradigms.*

  The shared memory architecture of Aquarius II makes it suitable for a wide variety of programming applications that exploit medium grain parallelism. This dissertation discussed the application of Aquarius II for logic programming. Its application for other parallel programming paradigms should be explored.

The multiprocessor simulation methodology employed in this dissertation has been invaluable in evaluating the performance of the proposed memory management scheme and the multiprocessor architecture. The modular design of NuSim makes it possible to simulate other multiprocessor architectures (i.e., processor, memory, and interconnection network) by replacing the appropriate modules. For example, the VLSI-PLM processor module may be replaced with a commercial microprocessor, or the cache lock state protocol may be replaced with another cache coherency protocol. The task kernel module for parallel Prolog may be replaced with parallel execution model for other programming paradigms. On the negative side, the interpretive nature of the simulator and the limited memory space and processing power of the host machine have somewhat restricted the full potential of the simulator. These problems can be reduced with more powerful host machines for simulation.

10.3 Concluding Remarks

As multiprocessors become more complex, the problem of memory management for parallel execution is increasingly difficult. It is often beyond the comprehension and the manageability of the programmer. Porting software to various multiprocessors while maintaining good memory performance is also a major problem. Therefore, to increase programmability of parallel architectures, memory management should be done by the system (and not by the application programmer).

Although shared memory multiprocessors provide an easy to program environment for a wide variety of applications, this shared space must still be properly managed. A programming language, such as Prolog, that provides implicit memory management support frees the programmer from the concern of memory management. This dissertation provides a possible implementation for such implicit memory management support. By combining two well known concepts (heap and stack), the resulting solution is more capable of adapting to the dynamic memory requirements of parallel execution. It is envisioned that other solutions to memory management problem will also be a hybrid of existing techniques to deal with the various levels of space requirement during parallel execution.

In the current state of computer technology, uniprocessor systems provide the lowest cost/performance ratio for most general purpose computing. Since multiprocessor systems take longer to build, they often do not take advantage of the latest processor technology. Parallel languages and programming environments need to sufficiently mature to take full advantage of the multiprocessors, while multiprocessor systems should become more widely available at much lower costs. As the programmers move away from the sequential programming mindset, the quantum leap in parallel processing may be realized: the development of practical and efficient parallel algorithms.

Appendix A

NuSim User's Manual

NAME

nusim – a multiprocessor simulation system for parallel execution of Prolog.

SYNOPSIS

nusim *-h*

nusim [*-option [option_argument] ...*] [*PLM-assembly-file*]

DESCRIPTION

NuSim is a simulator framework for the complete system simulation of a multiprocessor architecture: from the instruction set level to the memory architecture level with caches and communication protocols. The key feature of this simulator framework is flexibility, which allows for extensive instrumentation and continual updates and changes. The modular design identifies main features of the execution model and the architectures being simulated as cleanly separated modules with clearly defined interfaces. This allows for easy modifications to the individual modules to support new execution models and architectures.

NuSim's ease-of-use features include:

* **on-line help messages** to quickly show the default settings and briefly explain the commands. This also allows help messages to be updated more easily than being kept in a separate document.
* **comfirmation messages** to provide feedback that a command has been carried out properly or to explain the error if the command given is incorrect.
* **automatic initialization** by reading the commands from an initialization file upon starting up. This feature frees the user from having to repeatedly typing in the same commands upon initialization, such as which benchmark program to load and where to set the breakpoints.
* **a high level debugger**, called *NuSim Debugger*, which can interact with a symbolic debugger (such as *GNU GDB* or *Unix DBX*) to provide a multi-level level debugging environment.
* **a graphical interface**, called *xNuSim*, which provides a multiple window environment for viewing activities of processors and tasks.

Currently, the simulator supports the PPP Execution Model, which exploits AND/OR parallelism in Prolog programs, and a *Multi* memory architecture, multiple coherent caches on a single bus. The processor module of NuSim is the VLSI-PLM.

OPTIONS

The following options are available for configuring the multiprocessor system, the execution model, and the statistics collection:

-h	print the help message listing all the options and their current default values (in parentheses).
-s	toggle switch to simulate idle processors (NO)
-d	toggle interactive debug mode (DEBUG)
-i level	instrument to .data file 0.none 1.various 2.inst (1)
-m trace	memory traces to .mt file; trace: 0.no 1.data 2.code & data) (0)
-n	collect task (node) statistics (NO)
-p Procs	number of processors to simulate (max=8)
-t Tasks	number of tasks allowable (max=64)
-w Words	hashwindow size (128)
-q cycles	quantum between each .stat dump (5000000)
-u	set unordered output (ORDER)
-x cycle	time where execution is forced to terminate (MAX_INT)

The following are options to configure the cache system:

-Cd Kbytes	cache data size (64K)
-Cb Bytes	cache block size (16)
-Ca Assoc	cache set associativity (4)
-Cr Policy	cache replacement policy (l) l = LRU, f = FIFO, r = random
-Cw Policy	cache write policy (b) b = write back, t = write through
-Cs Integer	seed for random policy (1)
-Ci	enable instruction address tracing (off)
-CD	enable event stream output (off)

The following options are for configuring bus and memory latency:

-Ba cycles	bus arbitration time (1)
-Bb cycles	bus broadcast time (2)
-Ma cycles	memory access time (2)
-Mb cycles	memory burst time (1)
-Ms Kbytes	memory calloc() size (8K)

The following are the ELPS memory management options:

-Ep words	ELPS page size (4096)
-Es cycles	ELPS boundary overflow check time (2 for software; 0 for hardware)

NUSIM DEBUGGER COMMANDS

When NuSim running with the debugger option (ON by default) first starts up, commands from a file *nusim.startup* is executed, if this file exists. Then the prompt *NuSim:TOP>* will appear (after the initial 'run', subsequent breaks will show the prompt *NuSim:DBG>*. At the NuSim debugger prompt, 'h' for the root help menu. The commands at the root menu are:

h	print this help message
m menu	show the other menus: 1.system 2.display 3.breakpt 4.trace 5.dump
stat	show simulator status
load f	load file into code space
run	start simulation run
s	step simulation (single instruction)
c	continuous simulation
dbx	switch control to dbx debugger

Type 'm <menu #>' will show the other menus with corresponding explanations. The following are some of the commands used to set up breaktime, break points, and trace points:

be pid tid	set the break/trace environment, the processor and task pair for which the break/trace point is to take effect (-1 for all processors/tasks).
bt time	set the break time (cycle count of the simulated multiprocessor)
bp a 'b|t'	set a break/trace point at an address/label/procedure; the second argument 'b' or 't' to select whether to break or trace.
cbc n 'b|t'	change break/trace point #n for break or trace.
rm a	remove break/trace point at address/label/procedure.

rmall	remove all break/trace points
sb	show break/trace points

Tracing for all procedures/instructions can be done with the following commands:

trace lvf	set trace level: 0.off 1.instruction 2.procedure.
tl lv pid tid	set trace level for a specified processor and task; only one such trace can be set.
tp pid	set trace processor number
tt tid	set trace task number

For many other debugging commands, use the online menu system for help messages.

MULTILEVEL DEBUGGING

The NuSim debugger can interact with a C symbolic debugger (GNU GDB or Unix DBX) for debugging at both the VLSI-PLM instruction level and at the C code level (which represents the micro-engine). A dummy function *dbx_break()* is used to transfer control to the C symbolic debugger. To set up multilevel debugging, start up the C symbolic debugger and set up the breakpoint and alias as follows:

for DBX:

```
% dbx nusim
(dbx) stop in dbx_break
(dbx) alias menu "call debug_level()"
(dbx) alias c "cont"
(dbx) run [<nusim options> ... ] [PLM-assembly-file]
```

for GDB:

```
% gdb nusim
(gdb) break dbx_break
(gdb) define menu
    print debug_level()
    end
(gdb) run [<nusim options> ... ] [PLM-assembly-file]
```

After NuSim is invoked, the *NuSim:TOP>* prompt will appear. After the initial command 'run' to NuSim, subsequent breaks will show either the *NuSim:DBG>* prompt or the *(gdb)* (or *(dbx)*) prompt, depending on whether the breakpoint was set in the NuSim debugger or the C symbolic debugger. If the *NuSim:DBG>* prompt appears at the breakpoint, typing 'dbx' will get to the C debugger prompt, and typing 'c' will get back to the NuSim debugger before continuing execution. If the C debugger prompt is shown at the breakpoint, typing 'menu' will get to the NuSim debugger level, and multilevel debugging provides a simple way to observe data structures at the desired level of abstraction (a Prolog structure or a memory location) and setting breakpoints at the desired granularity.

LIMITATIONS

The size of memory that can be allocated for simulating the target multiprocessor memory is dependent on the swap space available on the host which executes the simulator.

SEE ALSO

xnusim(1)

AUTHORS

Tam M. Nguyen (simulation framework, debugger, and processor module),
Chien Chen (PPP task kernel), and
Mike Carlton (cache and memory module) –
University of California at Berkeley.

Bibliography

[ABY*87] S. Abe, T. Bandoh, S. Yamaguchi, K. Kurosawa, and K. Kiriyama. High Performance Integrated Prolog Processor IPP. In *Proceedings of 14th International Conference on Computer Architecture*, 1987.

[ACHS88] K. Appleby, M. Carlsson, S. Haridi, and D. Sahlin. Garbage Collection for Prolog Based on WAM. *Communications of the ACM*, 31(6), June 1988.

[AP84] O.P. Agrawal and A.V. Pohm. Cache Memory Systems for Multiprocessor Architectures. In *Proceedings of the 11th International Symposium on Computer Architecture*, Ann Arbor, MI, June 1984.

[Arc88] J. K. Archibald. A Cache Coherence Approach for Large Multiprocessor Systems. In *1988 ACM International Conference on Supercomputing*, ACM Press, Saint-Malo, France, July 1988.

[BBB*89] H. Benker, J.M. Beacco, S. Bescos, M. Dorochevsky, Th. Jeffre, A. Pohimann, J. Noye, B. Poterie, A. Sexton, J.C. Syre, O. Thibault, and G. Waltzlawik. KCM: A Knowledge Crunching Machine. In *16th International Symposium on Computer Architecture*, pages 186–194, May 1989.

[BCMD87a] W. Bush, G. Cheng, P. McGeer, and A. Despain. Experience with Prolog as a Hardware Specification Language. In *Proceedings of the 1987 IEEE Symposium on Logic Programming*, pages 490–498, San Francisco, September 1987.

[BCMD87b] W.R. Bush, G. Cheng, P.C. McGeer, and A.M. Despain. An Advanced Silicon Compiler in Prolog. In *Proceedings of the Intl. Conference on Computer Design*, pages 27 - 31, Oct. 1987.

[BD86] P. Bitar and A. Despain. Multiprocessor Cache Synchronization Issues, Innovations, Evolution. In *Proceedings of the 13th Intl. Symposium on Computer Architecture*, pages 424–433, Tokyo, Japan, June 1986.

[BdKH*88] U. Baron, J.C. de Kergommeaux, M. Hailperin, M. Ratcliffe, P. Robert, and J-C Syre. A Parallel ECRC Prolog System PEPSys: An overview and evaluation results. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, November 1988.

[BDL*88] R. Butler, T. Disz, E. Lusk, R. Olson, R. Overbeek, and R. Stevens. Scheduling for OR-Parallelism: An Argonne Perspective. In *5th Int'l Conference and Symposium on Logic Programming*, Seattle, Washington, August 1988.

[Bel85] C. G. Bell. Multis: a New Class of Multiprocessor Computers. *Science*, 228:462–467, April 16, 1985.

[BG87] J. Beer and W. Giloi. POPE - a Parallel-Operating Prolog Engine. 1987.

[BGW89] D.L. Black, A. Gupta, and W-D Weber. Competitive Management of Distributed Shared Memory. In *Spring COMPCON 89*, IEEE Computer Society Press, February 1989.

[Bit89a] P. Bitar. Automatic Program Annotation with SDDA. March 1989. Aquarius Parallel Model group discussion, UC Berkeley.

[Bit89b] P. Bitar. A Critique of Trace-Driven Simulation for Shared-Memory Multiprocessors. In *Proceedings of the Int'l Conference on Computer Architecture Workshop on Coherent Cache and Interconnect Structure for Multiprocessors*, Eilat, Israel, May 1989.

[BJCD87] M. Bruynooghe, G. Jenssens, A. Callebaut, and B. Demoen. Abstract Interpretation: Towards the Global Optimization of Prolog Programs. In *Proceedings of the 1987 Symposium on Logic Programming*, MIT Press, San Francisco, California, September 1987.

[BM82] C.G. Bell and J.E. McNamara. The PDP-8 Family. In D.P. Siewiorek, C.G. Bell, and A. Newell, editors, *Computer Structures: Principles and Examples*, pages 767–775, McGraw-Hill, 1982.

[Bor84] P. Borgwardt. Parallel Prolog Using Stack Segments on Shared-Memory Multiprocessors. In *Proceedings of the 1984 International Symposium on Logic Programming*, Atlantic City, NJ, Feb. 1984.

[BR86] P. Borgwardt and D. Rea. Distributed Semi-intelligent Backtracking for a Stack-based AND-parallel Prolog. In *IEEE 1986 Symposium on Logic Programming*, pages 211–222, Salt Lake City, Utah, September 1986.

[Bru82] M. Bruynooghe. The Memory Management of Prolog Implementations. In K.L. Clark and S.A. Tarnlund, editors, *Logic Programming*, pages 83–98, Academic Press, New York, NY, 1982.

[BSY88] R. Biswas, S.C. Su, and D. Yun. A Scalable Abstract Machine Model to Support Limited-OR (LOR)/Restricted-AND Parallelism (RAP) in Logic Programs. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*, Seattle, August 1988.

[Car89] M. Carlton. Cache Coherency for Multiple-Bus Multiprocessor architectures. March 1989. Technical Progress Report (November 1988 - March 1989), DARPA Contract No. N00014-88-K-0579.

[CC88] H. Coelho and J.C. Cotta. *Prolog by Example: How To Learn, Teach and Use It.* Springer-Verlag, 1988.

[CD90] M. Carlton and A. Despain. Cache Coherency for Multi-Multis. *submitted to Computer Magazine Nov 1989*, 1990.

[CG86] K. Clark and S. Gregory. PARLOG: Parallel Programming in Logic. *ACM Transactions on Programming Languages and Systems*, 8(1), January 1986.

[CGB89] D.R. Cheriton, H.A. Goosen, and P.D. Boyle. Multi-Level Shared Caching Techniques for Scalability in VMP-MC. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, Jerusalem, Israel, May 1989.

[CGS*85] W. Crowther, J. Goodhue, E. Starr, R. Thomas, and T. Blackadar. Performance Measurements on a 128-Node Butterfly Parallel Processor. In *Pro-*

ceedings of the 1985 International Conference on Parallel Processing, August 1985.

[CH83] A. Ciepielewski and S. Haridi. A Formal Model for OR-Parallel Execution of Logic Programs. In *Proc. Information Processing (IFIP) 83*, North-Holland, 1983.

[CH86] Andrzej Ciepielewski and Bogumil Hausman. Performance Evaluation of a Storage Model for OR-Parallel Execution of Logic Programs. In *Proceedings of the 3rd IEEE Symposium on Logic Programming*, pages 246–257, Salt Lake City, Utah, 1986.

[Cha85] J.H. Chang. *High Performance Execution of Prolog Programs Based on a Static Data Dependency Analysis*. PhD thesis, University of California at Berkeley, October 1985. CS Division Report No. UCB/CSD 86/263.

[CHN88] M. Carlton, B. Holmer, and T. Nguyen. MultiSim: A Complete Multiprocessor Cache Simulation System. May 1988. CS258 Class Report, CS Division, University of California at Berkeley.

[Cit88] W. Citrin. *Parallel Unification Scheduling in Prolog*. PhD thesis, CS Division, University of California at Berkeley, April 1988. Tech Report No. UCB/CSD 88/415.

[Clo85] W.F. Clocksin. Design and Simulation of a Sequential Prolog Machine. *New Generation Computing*, 3:101–103, 1985.

[Clo87] W.F. Clocksin. Logic Programming and Digital Circuit Analysis. *The Journal of Logic Programming*, 4:59–82, March 1987.

[CM87] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, New York, 3rd edition, 1987.

[Con83] J.S. Conery. *The AND/OR Process Model for Parallel Interpretation of Logic Programs*. PhD thesis, University of California, Irvine, June 1983. Technical Report 204.

[Con87] J.S. Conery. Binding Environments for Parallel Logic Programs in Nonshared Memory Multiprocessors. In *Proceedings of the 1987 IEEE Symposium on Logic Programming*, pages 457–467, San Francisco, September 1987.

[Cra85] J. Crammond. A Comparative Study of Unification Algorithms for OR-Parallel Execution of Logic Languages. *IEEE Transactions on computers*, C-34:911–917, October 1985.

[CVR86] W. Citrin and P. Van Roy. Compiling Prolog for the Berkeley PLM. In *Proceedings of the 19th Hawaii International Conference on System Sciences*, Honolulu, Hawaii, 1986.

[CVR88] Mike Carlton and Peter Van Roy. Distributed Prolog System with And Parallelism. *IEEE Software*, 5(1):43–51, January 1988.

[Dah88] V. Dahl. Representing Linguistic Knowledge Through Logic Programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*, MIT Press, Seattle, Washington, 1988.

[Deg84] D. Degroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, November 1984.

[DeG87] D. DeGroot. Restricted And-Parallelism and Side-Effects. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 80–89, San Francisco, August 1987.

[Dem82] J. Deminent. Experience with Multiprocessor Algorithms. *IEEE Transactions on Computers*, C-31(4), April 1982.

[DL87] T. Disz and E. Lusk. A Graphical Tool for Observing the Behavior of Parallel Logic Programs. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 46–53, San Francisco, August 1987.

[DLO87] T. Disz, E. Lusk, and R. Overbeek. Experiments with OR-Parallel Logic Programs. In *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987.

[Dob87a] T. Dobry. Xenologic's X-1. In *Proceedings of Spring Compcon*, San Francisco, CA, Feb. 1987.

[Dob87b] Tep Dobry. *A High Performance Architecture for Prolog*. PhD thesis, University of California, Berkeley, May 1987. Technical Report UCB/CSD 87/352.

[Doc88] T.W.G. Docker. SAME - A Structured Analysis Tool and its Implementation in Prolog. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*, MIT Press, Seattle, Washington, 1988.

[DPD84] T. Dobry, Y. Patt, and A.M. Despain. Design Decisions Influencing the Microarchitecture For A Prolog Machine. In *Proceedings of the MICRO 17*, October 1984.

[DS88] A. Despain and V. Srini. *Technical Progress Report: May 1988 - October 1988*. Technical Report, Computer Science Division, University of California, Berkeley, CA 94720, October 1988. DARPA Contract No. N00014-88-K-0579.

[EK87] S.J. Eggers and R.H. Katz. *A Characterization of Sharing in Parallel Programs and its Applicability to Coherency Protocol Evaluation*. Technical Report, CS Division, University of California, Berkeley, December 1987.

[Enc85] Encore Computer Corporation. Multimax Technical Summary. May 1985.

[End87] Endot, Inc. N-2 User Manuals. 1987.

[Fag87] B.S. Fagin. *A Parallel Execution Model for Prolog*. PhD thesis, University of California at Berkeley, November 1987. Technical Report UCB/CSD 87/380.

[FC88] R.M. Fujimoto and W.B. Campbell. Efficient Instruction Level Simulation of Computers. *Transactions of the Society for Computer Simulation*, 5(2):109–124, 1988.

[FD87] B.S. Fagin and A.M. Despain. Performance Studies of a Parallel Prolog Architecture. In *14th International Symposium on Computer Architecture*, June 1987.

[FM83] E.A. Feigenbaum and P. McCorduck. *The Fifth Generation: Artificial Intelligence and Japan's Computer Challenge to the World*. Addison-Wesley, 1983.

[FPSD85] B.S. Fagin, Y.N. Patt, V.P. Srini, and A.M. Despain. Compiling Prolog Into Microcode: A Case Study Using the NCR/32-000. In *Proceedings of the MICRO 18*, Asilomar, CA, December 1985.

[Fuj83a] R.M. Fujimoto. *SIMON: A Simulator of Multicomputer Networks.* Technical Report UCB/CSD 83/140, University of California at Berkeley, September 1983.

[Fuj83b] R.M. Fujimoto. *VLSI Communication Components for Multicomputer Networks.* PhD thesis, University of California at Berkeley, September 1983.

[Gea83] A. Gottlieb and et. al. The NYU Ultra Computer. *IEEE Transactions on Computers*, C-32, No. 2:175–189, February 1983.

[Geh87] E.F. Gehringer. *Parallel Processing: The Cm* Experience.* Digital Press, 1987.

[GF 85] G.F. Pfister, et al. The IBM Research Parallel Processor (RP3): Introduction and Architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, 1985.

[GMT89] A. Goto, A. Matsumoto, and E. Tick. Design and Performance of a Coherent Cache for Parallel Logic Programming Architectures. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, Israel, June 1989.

[Goo83] J. Goodman. Using Cache Memories to Reduce Processor-Memory Traffic. In *Proceedings of the 10th International Symposium on Computer Architecture*, Stockholm, Sweden, June 1983.

[GW88] J. R. Goodman and P. Woest. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii, June 1988.

[HB88] S. Haridi and P. Brand. ANDORRA Prolog – An Integration of Prolog and Committed Choice Languages. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 745–754, Tokyo, Japan, November 1988.

[HCH87] B. Hausman, A. Ciepielewski, and S. Haridi. Or-Parallel Prolog Made Efficient on Shared Memory. In *Proceedings of the 1987 IEEE Symposium on Logic Programming*, pages 69–79, San Francisco, September 1987.

[HEL*86] M. Hill, S. Eggers, J. Larus, G. Taylor, G. Adams, B.K. Bose, G. gibson, P. Hansen, J. Keller, S. Kong, C. Lee, D. Lee, J. Pendleton, S. Ritchie, D. Wood, B. Zorn, P. Hilfinger, D. Hodges, R. Katz, J. Ousterhout, and D. Patterson. Design Decisions in SPUR. *IEEE Computer*, :1 – 22, November 1986.

[Her86] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, University of Texas at Austin, August 1986. Department of Computer Sciences TR-86-20.

[HHS88] L. Hirschmann, W. C. Hopkins, and R. C. Smith. OR-Parallel Speed-Up in Natural Language Processing: A Case Study. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*, MIT Press, Seattle, Washington, 1988.

[Hil86] W.D. Hillis. *The Connection Machine.* MIT Press, Cambridge, Massachussettes, 1986.

[Hil87] M. Hill. *Aspects of Cache Memory and Instruction Buffer Performance.* PhD thesis, Ph. D. Thesis, University of California, Nov. 1987. CS Division Report No. UCB/CSD 87/381.

[HMSe86] J.P. Hayes, T. Mudge, Q.F. Stout, and et. al. A Microprocessor-based Hypercube Supercomputer. *IEEE Micro*, 6(5):6–17, October 1986.

[Hol88] Bruce Holmer. A Detailed Description of the VLSI-PLM Instruction Set. July 27, 1988. UC Berkeley CS Division Internal Report.

[HSC*90] B.K. Holmer, B. Sano, M. Carlton, P. Van Roy, R. Haygood, W.R. Bush, A.M. Despain, J. Pendleton, and T. Dobry. Fast Prolog with an Extended General Purpose Architecture. In *Proceedings of the 17th Intl. Symposium on Computer Architecture*, Seattle, Washington, May 1990.

[IH88] K. Iwanuma and M. Harao. Knowledge Representation and Inference Based on First-Order Modal Logic. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, November 1988.

[Int86] Intel Scientific Computers. Intel iPSC System Overview. 1986. Order Number 310610-001.

[JMP87] J.Gee, S. Melvin, and Y.N. Patt. Advantages of Implementing Prolog by Microprogramming a Host General Purpose Computer. In *Proceedings of the 4th International Conference on Logic Programming*, Melbourne, Australia, May, 1987.

[Jon86] D.W. Jones. Concurrent Simulation: An Alternative to Distributed Simulation. In *Proceedings of the 1986 Winter Simulation Conference*, pages 417–423, Washington D.C., December 1986.

[Kal87] L. V. Kale. The REDUCE-OR Process Model for Parallel Evaluation of Logic Programs. In *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987.

[KC87] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and Its Instruction Set. In *Proceedings of the 1987 Symposium on Logic Programming*, August 1987.

[KEW*85] R.H. Katz, S.J. Eggers, D.A. Wood, C.L. Perkins, and R.G. Sheldon. Implementing a Cache Consistency Protocol. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 276–283, Boston, June 1985.

[KRS88] L.V. Kale, B. Ramkumar, and W. Shu. A Memory Organization Independent Binding Environment for AND and OR Parallel Execution of Logic Programs. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*, Seattle, August 1988.

[KTW*86] Y. Kaneda, N. Tamura, K. Wada, H. Matsuda, S. Kuo, and S. Maekawa. Sequential Prolog Machine PEK. *New Generation Computing*, :51–86, April 1986.

[Kun82] H.T. Kung. Why Systolic Architectures? *IEEE Computer*, 15(1), January 1982.

[LBD*88] E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D.H.D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. The Aurora Or-Parallel Prolog System. In *Proceedings of the Int'l Conference on 5th Generation Computer Systems*, Tokyo, Japan, November 1988.

[Lin84] G. Lindstrom. OR-Parallelism on Applicative Architectures. In *Proc. 2nd Int'l Conf. on Logic Programming*, pages 159–170, Uppsala, July 1984.

[Lin88] Y.-J. Lin. *A Parallel Implementation of Logic Programs.* PhD thesis, University of Texas, Austin, August 1988. Technical Report AI88-84.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.

[LM86] P.P. Li and A.J. Martin. The Sync Model: A Parallel Execution Method for Logic Programming. In *IEEE 1986 Symposium on Logic Programming*, Salt Lake City, Utah, September 1986.

[LP84] G. Lindstrom and P. Panangden. Stream-Based Execution of Logic Programming. In *Proceedings of the 1984 International Symposium on Logic Programming*, pages 168–176, Atlantic City, NJ, February 1984.

[Mea83] J.R. McGraw and et. al. *SISAL: Streams and Iteration in a Single-Assignment Language.* Technical Report, Lawrence Livermore National Laboratory, 1983.

[Mel82] C. S. Mellish. An Alternative to Structure Sharing in the Implementation of a prolog Interpreter. In K.L. Clark and S.A. Tarnlund, editors, *Logic Programming*, pages 99–106, Academic Press, New York, NY, 1982.

[MU86] H. Mannila and E. Ukkonen. Timestamped Term Representation for Implementing Prolog. In *Proceedings of the 1986 Symposium on Logic Programming*, IEEE Computer Society, September 1986.

[Mud89] S. Mudambi. Performance of Aurora on a Switch-Based Multiprocessor. In E.L. Lusk and R.A. Overbeek, editors, *Logic Programming: Proceedings of the North American Conference*, MIT Press, 1989.

[Ngu90] T.M. Nguyen. *Complete Code Listing of the NuSim Multiprocessor Simulator.* Technical Report, CS Division, University of California, Berkeley, May 1990. (in preparation).

[NN87] H. Nakashima and K. Nakajima. Hardware Architecture of the Sequential Inference Machine: PSI-II. In *Proceedings of the 1987 Symposium on Logic Programming*, San Francisco, California, August 1987.

[NS88] T.M. Nguyen and V.P. Srini. A Two-Tier Memory Architecture for High Performance Multiprocessor Systems. In *1988 ACM International Conference on Supercomputing*, ACM Press, Saint-Malo, France, July 1988.

[Pan89] S.C. Pang. *xNuSim: Graphical Interface for a Multiprocessor Simulators.* Technical Report UCB CSD 89/532, CS Division, University of California, Berkeley, 1989.

[PBW85] E. Pittomvils, M. Bruynooghe, and Y.D. Willems. Towards a Real-Time Garbage Collector for Prolog. In *Proceedings of the 1985 Symposium on Logic Programming*, pages 185–198, Boston, MA, July 1985.

[PM86] R. Perron and C. Mundie. The Architecture of the Alliant FX/8 Computer. In A.G. Bell, editor, *IEEE Spring Compcon 86*, March 1986.

[PS87] F.C.N. Pereira and S.M. Shieber. *Prolog and Natural Language Analysis.* CSLI/SRI International, 1987. CSLI Lecture Notes Number 10.

[QSP85] J.S. Quarterman, A. Silberschatz, and J.L. Peterson. 4.2BSD and 4.3BSD as Examples of the UNIX System. *ACM Computing Survey*, 17(4), Dec 1985.

[Rei87] P.B. Reintjes. AUNT: A Universal Netlist Translator. In *Proceedings of the 1987 IEEE Symposium on Logic Programming*, September 1987.

[Rei88] P. B. Reintjes. A VLSI Design Environment in Prolog. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*, pages 70–81, Seattle, Washington, 1988.

[RF87] D.A. Reed and R.M. Fujimoto. *Multicomputer Networks: Message-Based Parallel Processing.* The MIT Press, Cambridge, Massachusetts, 1987.

[SA83] V.P. Srini and J.F. Asenjo. Analysis of Cray-1S Architecture. In *Proceedings of the 10th International Symposium on Computer Architecture*, pages 194–206, Stockholm, Sweden, June 1983.

[Sar88] R.G. Sargent. A Tutorial on Validation and Verification of Simulation Models. In *Proceedings of the 1988 Winter Simulation Conference*, December 1988.

[Sat80] M. Satyanarayanan. *Multiprocessors - A Comparative Study.* Prentice-Hall, Inc., 1980.

[SBN82] D.P. Siewiorek, C.G. Bell, and A. Newell. Personal Computing Systems. In D.P. Siewiorek, C.G. Bell, and A. Newell, editors, *Computer Structures: Principles and Examples*, pages 547–548, McGraw-Hill, 1982.

[Scr74] T.J. Scriber. *Simulation using GPSS.* John Wiley and Sons, New York, 1974.

[Sha86] Ehud Shapiro. Concurrent Prolog: A Progress Report. *IEEE Computer*, 19(8):44–58, August 1986.

[Shi88] T. Shintani. A Fast Prolog-Based Prodcution System KORE/IE. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*, MIT Press, Seattle, Washington, 1988.

[Sin88] Ashok Singhal. *PUP: An Architecture to Exploit Parallel Unification in Prolog.* Master's thesis, University of California at Berkeley, May 1988.

[Sin90] A. Singhal. *Exploiting Fine Grain Parallelism in Prolog.* PhD thesis, University of California, (expected June 1990).

[SKR88] W. Shu, L.V. Kale, and B. Ramkumar. *Implementation and Performance of Parallel Prolog Interpreter.* Technical Report Report No. UIUCDCS-R-88-1480, CS Dept., University of Illinois at Urbana-Champaign, December 1988.

[Smi82] A.J. Smith. Cache Memories. *Computing Surveys*, 14(3):473–530, September 1982.

[Sri88] V.P. Srini. A Low-Latency Crossbar Chip for Multiprocessors. *Patent Application, University of California*, Jan. 1988.

[Sri89] V.P. Srini. Crossbar-Multi-Processor Architecture. In *Proceedings of the Int'l Conference on Computer Architecture Workshop on Coherent Cache and Interconnect Structure for Multiprocessors*, Eilat, Israel, May 1989.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, 1986.

[STN*88] V.P. Srini, J. Tam, T. Nguyen, B. Holmer, Y. Patt, and A. Despain. *Design and Implementation of a CMOS Chip for Prolog*. Technical Report UCB/CSD 88/412, CS Division, UC Berkeley, March 1988.

[SW87] K. Shen and D.H.D. Warren. A Simulation Study of the Argonne Model for Or-Parallel Execution of Prolog. In *Proc. 1987 Symposium on Logic Programming*, August 1987.

[Sze89] P. Szeredi. Performance Analysis of the Aurora Or-Parallel Prolog System. In E.L. Lusk and R.A. Overbeek, editors, *Logic Programming: Proceedings of the North American Conference*, MIT Press, 1989.

[TD87] H. Touati and A. Despain. An Empirical Study of the Warren Abstract Machine. In *Proceedings of the 1987 IEEE Symposium on Logic Programming*, pages 114–124, San Francisco, September 1987.

[TGF88] S. Thakkar, P. Gifford, and G. Fielland. The Balance Multiprocessor System. *IEEE Micro*, 8(1), Feb. 1988.

[TH88] H. Touati and T. Hama. A Light-Weight Prolog Garbage Collector. In *International Conference on Fifth Generation Computer Systems 1988 (FGCS'88)*, Tokyo, Japan, 1988.

[Tic87] E. Tick. *Studies In Prolog Architectures*. PhD thesis, Stanford University, June 1987. Technical Report No. CSL-TR-87-329.

[TL87] P. Tinker and G. Lindstrom. A Performance-Oriented Design for OR-Parallel Logic Programming. In *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987.

[Ued85] Kazunori Ueda. *Guarded Horn Clauses.* Technical Report Technical Report TR-103, ICOT, June 1985.

[VR84] P. Van Roy. *A Prolog Compiler for the PLM.* Master's thesis, University of California, Berkeley, CA, August 1984.

[VR90] P. Van Roy. The Benefits of Global Flow Analysis for an Optimizing Prolog Compiler. *submitted to North American Conference on Logic Programming*, 1990.

[Wal85] S. Wallach. The Convex C-1 64-bit Supercomputer. In *Digest of Papers, Spring COMPCON 85*, pages 122–126, San Francisco, Feb. 1985.

[War83] D.H.D. Warren. *An Abstract Prolog Instruction Set.* Technical Report, SRI International, Menlo Park, CA, 1983.

[War84] D.S. Warren. Efficient Prolog Memory Management for Flexible Control Strategies. In *Proceedings of the 1984 International Symposium on Logic Programming*, pages 198–202, Atlantic City, NJ, February 1984.

[War87a] D.H.D. Warren. Or-Parallel Execution Models of Prolog. In *The 1987 International Joint Conference on Theory and Practice of Software Development (TAPSOFT '87 Proceedings II)*, pages 243–259, Springer-Verlag, Pisa, Italy, March 1987.

[War87b] D.H.D. Warren. The SRI Model for Or-Parallel Execution of Prolog - Abstract Design and Implementation Issues. In *Proceedings of the 1987 IEEE Symposium on Logic Programming*, pages 92–102, San Francisco, September 1987.

[Wei88] Allen Jia-Juin Wei. *DUES (Display Utilities and Environments for Simulation).* Master's thesis, Computer Science Division, University of California at Berkeley, 1988. Tech Report UCB/CSD 88/419.

[WHD88] R. Warren, M. Hermenegildo, and S.K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, MIT Press, Seattle, Washington, August 1988.

[Whi85] C. Whitby-Strevens. The Transputer. In *Proceedings of the 12th International Symposium on Computer Architecture*, June 1985.

[Wil87a] A.W. Wilson. Hierarchical Cache / Bus Architecture for Shared Memory Multiprocessors. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 244–252, June 1987.

[Wil87b] A.W. Wilson. Parallelization of an Event Driven Simulator for Computer Systems Simulation. *Simulation*, 49(2):72–78, August 1987.

[WMSW87] A. Walker, M. McCord, J.F. Sara, and W.G. Wilson. *Knowledge Systems and Prolog.* Addison-Wesley, 1987.

[WWS*89] P. Woodbury, A. Wilson, B. Shein, I. Gertner, P.Y. Chen, J. Barttlet, and Z. Aral. Shared Memory Multiprocessors: The Right Approach to Parallel Processing. In *Spring COMPCON 89*, IEEE Computer Society Press, February 1989.

[Yok84] M. Yokota. A Personal Sequential Inference Machine (PSI). *Proceedings of the International Workshop on Highlevel Computer Architecture-84*, May 1984.

[Zor89] B.G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms.* PhD thesis, University of California, December 1989. CS Division Report No. UCB/CSD 89/544.