# A CHARACTERIZATION OF PROLOG EXECUTION

by

MARK ANDREW FRIEDMAN

A thesis submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

(Computer Science)

at the

UNIVERSITY OF WISCONSIN-MADISON

1992

# CHAPTER 1

# Introduction

We believe Prolog to be an important foundation for future languages due to its very-high-level expressiveness, declarative style of programming, and inclusion of features not found in traditional, empirical programming languages. Prolog has proven useful in a variety of applications and has obtained a dedicated group of users in both research and in practice. Prolog will become increasingly useful as increasing numbers of symbolic applications emerge in future years.

The future of Prolog's success is dependent upon the existence of efficient Prolog implementations. Historically, Prolog implementations have been significantly less efficient (or perceived to be significantly less efficient) than necessary for practical use. Prolog's use of unification, backtracking and dynamic data structures create computational demands that continue to challenge the efficiency of its implementation. While in the past special-purpose Prolog machines, both complex microcoded machines and register-oriented, load-store instruction set architectures were designed that provided increased (but short-lived) Prolog performance, it is now difficult to justify the design of a Prolog-specific machine given that established load-store instruction set machines can provide Prolog performance of the same order of magnitude while serving general-purpose requirements as well. It is more difficult for the performance levels of language-specific machines to keep pace with the ever-increasing performance levels of general-purpose architectures that are driven by the momentum of procedural language implementations. Increasing the performance of Prolog systems is a continuing necessity to encourage the use of Prolog for applications which it is best suited.

It is the aim of our research to study the characteristics of Prolog execution and determine how these characteristics suggest improvements both at the abstract machine level and as modest modifications to established high-perfomance architectures to achieve increased Prolog performance with little impact on overall system performance. In particular, in this report we present a characterization of Prolog execution by measuring the occurrence and costs of a variety of operations on a simple load-store architecture using a new set of benchmarks. Driven by our characterization study, we propose enhancements to the architecture and quantify performance improvements that would be realized through the implementation of these enhancements.

We begin our discussion by expanding upon the significance of our problem and the directions and approach of our research. We describe the organization of the remainder of this report. We list original contributions that we make to Prolog implementation research.

### 1.1. Logic Programming and Prolog

Logic programming became the subject of increased attention when the Japanese Fifth Generation Computer Systems (FGCS) Project announced that Prolog would serve as the basis for its principal programming language [1]. The FGCS project succeeded in demonstrating that a framework of computer hardware and software could be constructed as a predicate logic machine. The achievements of FGCS include a hierarchy of logic programming languages, the development of a logic programming operating system, and the development of a family of logic programming machines [2-4]. The FGCS project stimulated similar programs in the United States [5-7] and created interest in logic programming throughout the world.

Logic programming is a programming paradigm founded on simple, first-order logic, yet it is powerful enough to facilitate the development of advanced applications [8-10]. This idea was first explored by Kowalski and Colmerauer in the early 1970's [11, 12]. Prolog is the most popular logic programming language. Introductions are provided in tutorials by Clocksin and Mellish [13], and Sterling and Shapiro [14], and in a pictorial overview by Colmerauer [15]. The language is especially suited for symbolic programming in natural language analysis, artificial intelligence, and database applications, as well as for applications such as compiler writing [16], algebraic manipulation, and theorem proving. Successful examples of Prolog's use are in natural language understanding [17-20], expert systems [21-23], knowledge representation [24], the automatic generation of plans [25], database design [26, 27], expert database systems [28-30], and computer aided design (CAD) [31-34].

Programming in Prolog is distinguished from programming in conventional languages in that Prolog is a declarative language. Prolog programmers describe known facts and relationships within the domain of the problem they wish to solve, together with a query defining the problem. Programmers are freed from the task of prescribing the exact sequence of operations needed to solve a problem as they must do in imperative languages. A more accurate description of the activity of "programming" in Prolog is "the development of logical specifications of the knowledge within a particular domain that is necessary to solve a problem".

When a computer carries out the computation of a Prolog program, the sequencing of the computation is determined by the internal inference mechanisms of the Prolog system. Prolog searches through the originally supplied set of facts and relationships and applies them to attempt to solve a problem. This ability to automatically support the computation process is based on the principles of unification and resolution discovered by Robinson [35].

Resolution is a rule of inference of first-order predicate logic. It can be used to determine how one formula (in Prolog's case, a conjunction of subproblems of the original problem) is logically derived from a set of other formulae (in Prolog's case, a larger problem to be solved and the facts and relationships supplied by the programmer). Prolog uses resolution to break the originally defined problem into smaller subproblems, and these subproblems into still smaller subproblems, until the system arrives at a set of subproblems for which answers can be obtained immediately from the set of programmer-supplied facts.

Unification is a generalized pattern-matching mechanism applied to a set of formulae that constructs more specific versions of these formulae so that resolution can be applied. During unification, terms within the formulae are matched and made "identical" by instantiating variables within the terms to specific values. It is within this process that data structures are created as a solution is obtained to the program's problem.

An introduction to first-order predicate calculus and resolution-based theorem proving can be found in Nilsson [36]. Mathematical questions regarding the properties of Prolog's procedural meaning with respect to logic are analyzed by Lloyd [37]. In the remainder of this thesis, we refrain from discussing theoretical aspects of logic programming.

The characteristic that gives Prolog it's primary advantages over procedural languages is the separation between its logical and control components. This characteristic results in its declarative nature and a separation of the two chief concerns during program development: the correctness and the efficiency of a program. The Prolog programmer is better able to address the concerns of correctness and verification because the concerns of efficiency are lower-level details left to the implementer of the Prolog system. The powerful pattern-matching scheme of unification gives Prolog the ability to easily create, compare, search, and manipulate complex data structures.

Declarativeness, combined with the ease of use of arbitrary data structures, results in a very-high-level programming language where the programmer is able to express ideas in terms of objects and entities in the language of his problem's domain. The relative independence of the facts and relationships that the programmer supplies makes Prolog well suited for incremental development and program maintenance. All of these features combine with the conciseness of Prolog to make it an ideal language for rapid prototyping. In all, Prolog is an important foundation for future languages.

Unfortunately, the earliest Prolog systems, implemented as interpreters [11], executed slowly and required unacceptable amounts of working storage. The unavailability of fast computers with large main memories and the unavailability of optimizing compilers hindered the establishment of Prolog as a practical language. The development of Prolog was slowed by an absence of interesting examples to demonstrate its novelty and the existence of better compilers and environments for the then more mature language Lisp that had proven its value in symbolic processing. The development of the first Prolog compiler by Warren [38] and his later refined definition of an abstract Prolog instruction set and memory organization [39] provided the first steps toward efficient Prolog implementation. Several research efforts followed exploring the design of special-purpose, high-performance processors that directly implemented the instruction set of Warren's abstract machine [40-42]. As the trends of conventional language architecture design moved to that of reduced-instruction-set-computer (RISC) technology, Prolog architecture researchers looked to improve Prolog through RISC implementation techniques [43, 6, 44-48]. Recently, researchers have begun investigating techniques of global optimization specifically suited for Prolog compilation [49-53]. The most recent and promising Prolog implementation, the Berkeley Abstract Machine (BAM) [7], coupled with the Aquarius compiler [54, 55], combines a RISC architecture with a global optimizing compiler.

Despite advancements in the development of Prolog systems, the current techniques employed to implement the language and more specifically, the effects of the implementation of dynamic data structures, dynamic typing, searching mechanisms, and the process of unification create heavier computational demands than desirable (that is, long execution times and large space requirements when compared with other languages). The characteristics of Prolog create computational needs different from other languages. (Dynamic data structures and dynamic typing are features not commonly found in procedural languages. Searching mechanisms and the process of unification are not built into either procedural languages nor functional languages.) It is relatively recently that researchers in the fields of computer architecture and compilation theory have joined researchers in logic programming to address these unique requirements. We expect that with continued research such as the work we present in this thesis that Prolog systems will offer performance comparable to other languages making logic programming languages the definitive choice for those applications that they suit best.

### 1.2. Overview of Research

Our research centers on discovering and quantifying implementation improvements that lead to more efficient processing of programs written in Prolog. Our work has been in three directions. First, we performed a comprehensive, bottom-up characterization of Prolog programs by exploring the frequency of occurrence of different operations and computing the costs of these operations as the count of instructions executed on a simple load-store architecture for a set of benchmarks representative of typical Prolog programs. Our analysis provides a foundation for the development of future Prolog systems. It is more complete (in terms of the quality of benchmarks, the amount and presentation of fundamental statistics, and the precision of cost calculations) than any work that has been presented of which we are aware. We compare our measurements to Prolog profiles reported by others [5, 56, 57] and to studies of other languages [58-61] when these comparisons lend insight towards implementing Prolog efficiently.

The study of operational characteristics of programs written in high-level programming languages leads to information that can be used to detect inefficiencies in the implementation of programming languages and suggest possible improvements in their design. The second portion of our work identifies issues related to the abstract machine model and to computer architecture that are most critical for efficient Prolog execution by drawing from our characterization of Prolog. We propose or reference alternate implementation and enhancement techniques for these issues. We present comparative performance analyses to quantify the improvements gained through these techniques by our benchmarks. Our results are useful to compiler and interpreter designers who wish to implement Prolog on an existing machine and to architects who are looking to add hardware support to improve Prolog performance especially on modern load-store architectures.

The final portion of our work is the beginning of future research into exploiting instruction-level parallelism. We have measured the parallelism and speed-up that would be achieved in implementations of Prolog using a simple pipelined machine model and using a simple multiple-operation-issue machine model. We propose new directions to explore that may allow Prolog to fully exploit the increasing levels of instruction level parallelism attainable in modern pipelined and multiple-operation-issue architectures.

Our results are obtained using a new set of eleven benchmarks that include examples from artificial intelligence, natural language analysis, text processing, compiler technology, database technology, number theory, and linear algebra. The benchmarks we have developed are considerably larger and more representative of real applications than benchmarks that predominantly appear in the literature [38, 5, 7].

We have developed the Simple Instruction Set Machine for Prolog Execution (SIMPLE) to generate our results. The SIMPLE system includes an architectural specification, a compiler, and a simulator. The front-end of our compiler is the Programmed Logic Machine (PLM) compiler developed by Van Roy at Berkeley [62] that translates Prolog code into a representation of the Warren Abstract Machine (WAM). The back-end of the SIMPLE compiler implements the WAM model using the instruction set, register set, and memory organization of the SIMPLE architecture. The simulator emulates SIMPLE instructions and generates an extensive set of static and dynamic statistics and instruction and memory reference traces.

SIMPLE is a load-store, register-oriented instruction set architecture. Only load and store operations access memory. Operands of other instructions are immediate values or are contained within registers. The SIMPLE architecture is similar to many modern register-oriented computers [44-46] in that the operations and addressing modes within its small instruction set are the single-cycle, fixed-size, fixed-format instructions that form the core of these machines. Our work supports the philosophy of reduced-instruction-set-computer (RISC) design in that we have selected a minimal set of essential instructions suitable for fast hardware execution as our architectural base. Suggested enhancements involve small modifications to the instruction set that are justified by their frequency of use and non-disruptive to the overall performance and optimizations at the abstract machine level that lend synergy to the architecture. The RISC approach has been very beneficial to conventional architecture design [63, 64]. As we have seen in recent Prolog developments [6, 7], application of similar design techniques targeted to Prolog lead to results that are as profitable. Because our architecture is similar to existing machines and because its instructions are easy to understand, our results are readily applied to the design of other machines.

### 1.3. Organization of Thesis

The organization of the remainder of this thesis is as follows. In chapter 2, we provide background information on other researchers' work on implementations of Prolog dividing these efforts into five areas: Warren's abstract machine model of Prolog execution, special-purpose Prolog machines, reduced-instruction-set-computer implementations of Prolog, global optimizations for Prolog compilation, and the most recent development, the Berkeley Abstract Machine [7]. We briefly overview the Warren Abstract Machine [39]. We survey Tick's overlapped Prolog processor [65], the Berkeley Programmed Logic Machine (PLM) [41], and the Fifth Generation Computer Systems Project's Personal Sequential Inference machines (the PSI-I and PSI-II) [66, 42]. We review the work that generated interest in Prolog RISC architectures by the Symbolic Processing Using RISC (SPUR) group [43] and the LOW RISC group [6]. We discuss global analysis compiler techniques on automatic mode determination [49, 51] and on automatic detection of determinism [50, 52]. We review the Aquarius compiler [55] and the Berkeley Abstract Machine (BAM) processor [7],

Chapter 3 contains a presentation of our research methodology, tools and benchmarks. We describe the SIMPLE architecture, the organization of the SIMPLE compiler, and the use of SIMPLE to emulate the Warren Abstract Machine. We discuss the use of the SIMPLE simulator in obtaining statistics and the compilation of our measurements into results. We introduce our benchmark suite and characterize their static code characteristics, dynamic code characteristics, and memory usage.

The results of our characterization study, and a discussion and analysis of techniques to enhance critical aspects of a Prolog implementation are contained in chapters 4 and 5 separated into a WAM-level profile and an architectural-level study. Chapter 4 examines the costs of executing WAM operations and the costs created by Prolog's built-in predicate routines and evaluates the costs of operations fundamental to a WAM implementation. Chapter 5 reports on architectural operations including higher-level operation frequencies (for example, tag handling, procedure calls, and accessing different memory areas) and lower-level architectural statistics (for example, instruction frequencies, distribution of operand and addressing use, and patterns of memory usage).

As benchmark characteristics are presented and important issues are identified, we present performance analyses of alternate implementation techniques. We discuss architectural support for tag-handling, performance improvements through compile-time mode and type determination, elimination of redundant dereferencing operations, in-line expansion of fundamental operations and built-in predicate routines, evaluation of arithmetic expressions, support of last-in, first-out stacks, and the representation of fact-intensive programs.

Chapter 6 presents our preliminary instruction-level parallelism results that form the basis for future work. We report the instruction-level parallelism and speed-up obtained from executing our benchmarks on a pipelined machine model and on a multiple-operation-issue machine model and propose directions for future exploration of performance improvements.

Prolog architectural research has evolved from the design of special-purpose WAM instruction set machines to the tuning of register-oriented, general-purpose architectures coupled with compilers that expose and optimize the inner details of WAM operations through global and special-case analysis. In chapter 7, we conclude that our work advances this trend that has proven beneficial. By analyzing architectural-level details to identify striking or noticeably different characteristics of a benchmarked simulation, one can propose and quantify improvements for modest enhancements to an architecture and adjustments to the abstract machine model to add significant support for Prolog on general-purpose, load-store instruction set machines. The need for such analysis continues as general-purpose architectures involve and Prolog systems strive to compete with conventional language performance by exploiting new enhancements. We suggest techniques directed at exposing instruction-level parallelism as an area for future work using our approach.

We include one appendix. Appendix A describes a technique to reduce the very large static SIMPLE code size of fact-intensive programs with a modest decrease in performance. While this technique relates to our overall work, it does not fit neatly into the flow of the main body of this report.

In the remainder of this report, we assume the reader is familiar with Prolog: the terminology and the use of the language. We assume the reader is familiar with Prolog's inference mechanism, specifically the methods of unification and resolution and their application in the depth-first, left-to-right goal selection search carried out by Prolog (SLD-resolution) and implemented through backtracking. An appreciation of how an interpreter executes a Prolog program would be sufficient depth for understanding our presentation. For a review of this material and the terminology that we use, the reader is referred to Clocksin and Mellish [13].

### 1.4. Original Contributions

The main original contributions that we add to research in Prolog implementation are as follows:

(1) Our characterization study is of value to a wide range of researchers including those interested in Prolog abstract machine models, Prolog compiler design, and Prolog architectures. Other researchers have used similar types of information in their work. We know of no single report that characterizes Prolog as completely, that covers as many viewpoints, and that utilizes as interesting a set of benchmarks. Furthermore, no research of that we are aware has reported the characteristics of Prolog from the viewpoints of locality of code, non-executed code, high-level architectural operations, and the use and properties of built-in predicate routines.

(2) We introduce a new set of Prolog benchmarks that are larger and more realistic than benchmarks that have appeared in the literature.

(3) We introduce tag-handling support to our architecture altering the semantics of the instruction set so that the architecture recognizes that Prolog objects consist of both a tag field and a value field. Research in the design of the Berkeley Abstract Machine (BAM) architecture has suggested similar instructions to improve performance [7].

(4) We investigate the behavior of our benchmarks to determine how much improvement may be achieved through global analysis compiler techniques for a variety of optimizations. Mellish [49], and Debray and Warren [51] have developed methods that automatically determine mode information and determinism. We analyze no specific techniques but instead use a reverse engineering scheme to determine the potential performance improvements that can be achieved through such techniques for specific benchmarks.

(5) We recognize the importance and costs of built-in predicates and observe our benchmarks' behavior to determine the maximum benefits that could be achieved in these operations from global optimization techniques. Specifically, we suggest improvements in the compilation of built-in math predicate routines to increase performance and decrease memory requirements.

(6) Our study of instruction-level parallelism begins work in determining what low-level parallelism is available in the execution of Prolog programs. There have been no reported studies of this type specific to Prolog.

(7) We propose a combined WAM-database model of execution to significantly reduce the size of fact-intensive benchmarks resulting in only a modest increase of executed instructions.

Our results have been generated through our own work. We have developed all of our own tools with the exception of the front-end of our compiler. A significant amount of time was spent developing the back-end of the compiler and optimizing the sequences of SIMPLE instructions that emulate WAM operations. A significant amount of time was spent developing the simulator that includes debugging and statistic generation capabilities. We spent a significant amount of time gathering and modifying our benchmark code to produce a diverse and realistic set of examples.

# CHAPTER 2

# Background

## 2.1. The Warren Abstract Machine

Prolog originated in the 1970's through efforts by Robert Kowalski and his expertise in logic and automatic theorem proving [11], Alain Colmerauer and his interests in natural language processing [12], and Phillipe Roussel and his work in implementing Prolog systems. Early Prolog interpreters were not favorably received due to their slowness and large storage requirements. It was David H. D. Warren's thesis work developing a Prolog compiler that led to the acceptance of Prolog and established the basis for practical systems [38, 67]. Warren refined his ideas and developed an "abstract Prolog engine". Known as the Warren Abstract Machine (WAM) [39], this model has become the standard for implementing Prolog systems. The Warren Abstract Machine defines an abstract instruction set and memory organization. The WAM description which follows includes details and enhancements to Warren's original definition which were incorporated as part of the Programmed Logic Machine (PLM) [41].

In the WAM, each predicate of a Prolog program is compiled into a single procedure. A Prolog goal is interpreted as a procedure call with the arguments of the goal serving as the arguments of the procedure call. The execution of a goal is the execution of the procedure of the predicate determined by the goal's principle functor and arity.

A procedure contains code for each of its predicate's clauses. The head of each clause becomes an entry point into the procedure. The code for each clause executes a series of tests and assignments to unify the arguments passed to the procedure with the terms and variables defined in the head of the clause. If the unification of all arguments succeed, a sequence of procedure calls then follow to execute the subgoals in the clause's body. If each of the subgoals executes successfully, the procedure returns to its caller.

If unification of the procedure arguments fails, or if all alternate clauses of a subgoal of the procedure fail, a backtracking routine is initiated which considers the next clause of the procedure. Each procedure contains code to assist the backtracking mechanism. When a procedure is first called, the arguments passed to the procedure are saved before other code is executed. Also saved, is the next alternate clause to be tried should backtracking become necessary. When backtracking occurs, the procedure arguments are restored to their original values and the next alternate clause is executed. The alternate clause information must also be updated to the clause which follows the newly selected clause. When the last clause of a procedure is to be executed, backtracking information is no longer needed and is discarded.

### 2.1.1. Data Objects

In the WAM, a Prolog object is represented by a word containing a tag and a value. The tag distinguishes the type of an object. The main types are unbound variables, references (or bound variables), constants, structures, and lists. An object's tag field assists in cdr-coding and garbage collection.

Figure 2.1 shows typical representations for these data objects. Both variables and references contain a variable tag. The value of a variable is a memory reference to itself while the value of a reference addresses the object to which it refers. Constants contain a constant tag, a secondary tag to distinguish between atoms, integers, and nil, and an immediate value or symbol table address. Structures and lists contain an appropriate tag and a pointer to the first of their sequence of elements. For structures, the first element of this sequence is the constant specifying its functor. The structure's arguments occupy successive locations in memory terminated by nil. For lists, elements may occupy consecutive memory locations or the list may be broken in pieces which are chained through cdr-coding [41]. In cdr-coding, a location which contains a list tag with the special cdr-bit set to one serves as a continuation pointer to the next piece of the list. A list is terminated by nil. A list may also be "non-terminated" by a variable element containing a set cdr-bit. Such a list may continue to grow as the execution of a program proceeds.

### 2.1.2. Memory Areas

The WAM contains five memory areas divided among a code and data space. The code space contains instructions and other information representing the program. The four data space areas are accessed as stacks. These are the local stack, the global stack, the trail, and the push-down list (PDL). These stacks generally grow with each procedure invocation and contract on procedure return or on backtracking.

The global stack contains all the structures and lists created by unification and procedure invocation. It is analogous to the heap in procedural languages.

| | Address | Tag | Value |
|---|---|---|---|
| variable: `X` | 100 | variable | 100 |
| reference to next object | 200 | variable | 300 |
| constant: `32` | 300 | constant (integer) | 32 |
| structure: `foo(Y)` | 400 | structure | 450 |
| | 450 | constant (atom) | “foo” |
| | 451 | variable | 451 |
| | 452 | constant (nil) | nil |
| list: `[123,bar,321]` | 500 | list | 550 |
| | 550 | constant (integer) | 123 |
| | 551 | constant (atom) | “bar” |
| | 552 | list (cdr set) | 570 |
| | 570 | constant (integer) | 321 |
| | 571 | constant (nil) | nil |
| non-terminated list: `[foobar\|X]` | 600 | list | 650 |
| | 650 | constant (atom) | “foobar” |
| | 651 | variable (cdr set) | 651 |

**Figure 2.1. WAM Data Objects**

The local stack contains environments and choice points. Environments hold storage and control information for individual clause invocations. Variables within a clause are classified as either permanent or temporary. Permanent variables are those which must survive across procedure calls. The permanent variables of a clause are stored in its environment. Environments also contain a “continuation” which holds information specifying how to proceed after the execution of the clause completes. Environments correspond to the procedure activation records of procedural languages.

Choice points contain the information necessary to return to an earlier state of computation when backtracking. When a procedure is entered which contains more than one clause to be tried, a choice point is created. The choice point contains the original values of the procedure arguments, the address of the next alternate clause, and the values of several pointers to memory described in the next section. Choice points are unique to logic programming languages. The deterministic nature of imperative languages eliminates the need for an equivalent structure.

The trail area contains locations of variables which have been bound during unification and which must be unbound on backtracking to restore a previous state. This area is also unique to logic programming languages.

The push-down list is a small scratch area which assists in unification and in the implementation of built-in predicate routines.

### 2.1.3. Address Pointers

The state of a Prolog computation is defined by a set of address pointers to memory. These include a program counter (regP), a continuation program pointer (regCP), pointers to the current environment (regE) and the current choice point (regB), pointers to the top of the global stack (regH), the top of the trail stack (regTR), and the top of the push-down list (regPDL), and a pointer to the global stack backtrack point (regHB). The state of the computation also includes the current procedure argument values and the values of the procedure's temporary variables. Typically, WAM implementations share a single set of resources to store these values (reg1 - reg8).

Figure 2.2 shows the layout for the WAM memory areas, the directions in which the stacks grow, and the WAM address pointers into each area.

In figure 2.3 the information contained in an environment is displayed which includes the previous contents of WAM address pointers and storage for permanent variables (Y1 - YN).

Figure 2.4 shows the information which is saved within a choice point. This includes the procedure arguments and temporary variable values, WAM address pointers, and the alternate clause address.

### 2.1.4. WAM Operations

Prolog programs are encoded as sequences of instructions which execute WAM operations. Generally, compilation translates each symbol of a Prolog program (for example, constants, variables, functors, and operators) into a

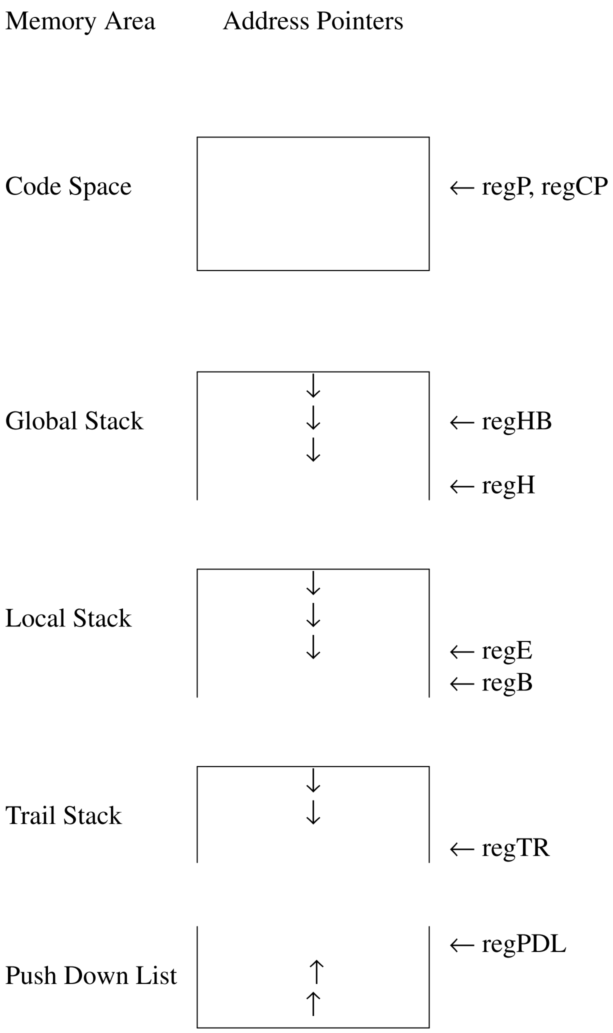


**Figure 2.2. WAM Memory Areas**

| | |
|---|---|
| old regE | ← regE |
| old regB | |
| old regCP | |
| old regN | |
| Y1 | |
| . . . | |
| YN | |

**Figure 2.3. WAM Environment**

| | |
|---|---|
| saved regB | |
| “alternate clause” | |
| saved regCP | |
| saved regE | |
| saved regN | |
| saved regH | |
| saved regTR | |
| saved reg1 | |
| . . . | |
| saved reg8 | |
| | ← regB |

**Figure 2.4. WAM Choice Point**

WAM operation. WAM operations, summarized in table 2.1, can be classified into compiled-unification, procedural, choice point, and switch operations. Each clause of a program is constructed from compiled-unification and procedural operations. Clauses are linked into procedures with choice point and switch operations. Procedural operations accomplish the sequencing between Prolog procedures.

Unification Operations

| | |
|---|---|
| get_variable Xi,Aj | get_variable Yi,Aj |
| get_value Xi,Aj | get_value Yi,Aj |
| get_integer I,Aj | get_atom S,Aj |
| get_nil Aj | |
| get_structure F,Aj | get_list Aj |
| | |
| put_variable Xi,Aj | put_variable Yi,Aj |
| put_value Xi,Aj | put_value Yi,Aj |
| put_integer I,Aj | put_atom S,Aj |
| put_nil Aj | |
| put_structure F,Aj | put_list Aj |
| put_unsafe_value Yi,Aj | |
| | |
| unify_void N | |
| unify_variable Xi | unify_variable Yi |
| unify_value Xi | unify_value Yi |
| unify_integer I | unify_atom S |
| unify_nil | |
| unify_cdr Xi | unify_cdr Yi |
| unify_unsafe_value Yi | |

Procedural Operations

| | |
|---|---|
| call P,N | execute P |
| proceed | |
| escape B | |
| allocate | deallocate |

**Table 2.1. WAM Operations**

Choice Point Operations

| | |
|---|---|
| `try_me_else C` | `retry_me_else C` |
| `trust_me_else fail` | |
| `try C` | `retry C` |
| `trust C` | |
| `cut` | `cutd C` |

Switch Operations

| | |
|---|---|
| `switch_on_term Cc,Cl,Cs` | |
| `switch_on_constant N1,N2` | `switch_on_structure N1,N2` |

**Table 2.1. (continued)**

The compiled-unification operations include get, put, and unify operations. The get operations correspond to the terms and variables in the head of a clause. They are used to retrieve and unify the procedure's arguments with the structure of the clause head. The type of each term in the head determines the type of the corresponding get operation. Variations of get operations match permanent and temporary variables, constants, structures, and lists. Get operations may simply copy an argument value to a permanent or temporary variable (`get_variable`). They may unify an argument value with a permanent or temporary variable (`get_value`). They may verify that an argument has a specific type and a specific value or may assign a specific type and specific value to an uninstantiated variable (`get_integer`, `get_atom`, `get_structure`, `get_list`, `get_nil`).

The put operations correspond to the arguments of a subgoal. They are used to create arguments in preparation for a procedure call. Put operations create permanent and temporary variables (`put_variable`), duplicate references to existing objects (`put_value`), and create constants, structures, and lists (`put_integer`, `put_atom`, `put_structure`, `put_list`, `put_nil`).

The unify operations combine get and put functionality by both unifying arguments passed into a procedure and creating arguments for a subsequent procedure call. Unify operations correspond to the elements of structures and lists. A sequence of unify operations is preceded by either a get or put operation for a list or structure. This proceeding operation determines the mode in which the following unify operations will execute. In “read mode”, unify operations perform unification with successive elements of an existing list or structure. In “write mode”, unify operations create the successive elements of a new list or structure. As with get and put operations, the unify operation is selected by the type of the corresponding term in the Prolog clause.

The procedural operations correspond to the predicates which form the head and subgoals of a clause. Procedural operations transfer control to the subgoals (`call`, `execute`) and return control to a parent when a subgoal succeeds (`proceed`). Procedural operations manage environments by allocating space on the local stack for the permanent variables of a clause (`allocate`) and deallocating space (`deallocate`) when the execution of a clause completes.

Choice point operations link together the clauses of a procedure to determine the sequence of clauses which are selected when attempting to solve a goal. Each clause of a procedure is preceded by a choice point operation. The first choice point operation is responsible for creating a choice point (`try_me_else`). Intermediate clauses are preceded by choice point operations which update the alternate clause address (`retry_me_else`). The choice point operation before the last clause removes the choice point of that procedure (`trust_me_else`).

Switch operations speed up Prolog by determining the set of clauses which could potentially match the arguments passed into a procedure. These instructions branch to an address of a single potential clause or to a sequence of calls to potential clauses (`try`, `retry`, `trust`) based on either the type of the first argument (`switch_on_term`) or the value of the first argument (`switch_on_constant`, `switch_on_structure`).

#### 2.1.5. Fundamental Operations

Several implicit operations have significant importance and high frequency of use and are considered fundamental WAM operations. These are the bind, trail, dereference, fail, detrail, and general unification operations. In the PLM, which uses cdr-coding, another fundamental operation is the decdr operation. The bind operation performs instantiation. It converts variables to references and other objects. The trail operation is performed when a variable is bound. It manages the trail by pushing variable addresses onto the trail stack which must be unbound when backtracking. Not all variables which are bound need to be “trailed”. Trailing only occurs for global stack variables above the global stack backtrack address and for permanent variables above the local stack backtrack address. This check is included in the trail operation. The dereference operation is used to follow chains of reference pointers to their eventual value. These chains are created during the binding of variables to other variables.

The dereference operation is used within compiled unification and switch operations, within general unification, evaluate, decdr, and within many of the built-in predicate routines. The fail operation implements backtracking. It restores the machine state from the current choice point and uses the detrail operation to reset trailed variables to unbound. The fail operation then transfers control to the most recent alternate clause. The general-unification operation unifies arbitrary Prolog objects and binds variables within these objects as required. It is used within several of the WAM operations (`get_value`, `unify_value`). General unification uses the push-down list as a scratch pad for unifying nested lists and structures. The decdr operation is used while unifying lists and structures to obtain the next element to be processed in these objects. It must examine the cdr-bit of list elements to properly follow list-continuation pointers.

#### 2.1.6. Built-In Predicates

An implementation modeled after the WAM must integrate a library of built-in predicates (also known as evaluable predicates) into the Prolog system. These predicates either cannot be defined in "pure" Prolog or they are used often enough that it is desirable to build them into the system. Many of the built-in predicates are metalogical or extralogical operations which result in side effects. The core of these predicates is common to most Prolog systems.

A few built-in predicates can be implemented using WAM operations (`not`, `repeat`, `fail`). Other built-in predicates are implemented through explicit operations or instructions (`cut`). Most WAM implementations supply built-in predicates through an escape mechanism. This escape mechanism may be a procedure call to a routine to accomplish the operation of the built-in predicate or may be a remote procedure call to do the same on another processor.

The evaluate operation which determines the value of arithmetic expressions and is used in the math built-in predicate routines is another fundamental WAM operation. In Prolog, arithmetic expressions are not immediately evaluated. Instead, they are represented by structures with functors corresponding to arithmetic operators and arguments which are themselves arithmetic expressions or numerical constants. In the evaluate operation, the form and functor of an expression is examined. Its arguments are extracted. An arithmetic operation is performed within the unify operations to obtain a numeric value. In a complex expression where the arguments of the expression are also expressions, the arguments must first be evaluated before the value of the top-most expression can be completed. The push-down list is used as a stack to evaluate nested expressions.

### 2.1.7. Palindrome Example

Consider the palindrome Prolog program of figure 2.5. This program takes an original list, reverses it, and verifies that the reversed list is equivalent to the original list. In figure 2.6, we show the WAM operations for the palindrome example. The palindrome WAM code consists of WAM operations for each predicate and for each clause of the program. The sequence for each clause starts with a get operation for each of the terms in the head of the clause. When the get operation corresponds to a list, it is followed by unify operations for each of the elements in the list. The get and unify operations retrieve arguments to match with terms of the clause head and assign argument terms and subterms to permanent and temporary variables. These unification operations are followed by a group of put and unify operations and concluded by a procedural operation for each of the subgoals in the clause body. There is a put operation for each of the arguments of the subgoal which will be called. For list arguments, unify operations are used for each of the elements of the list that is passed to the subgoal. The put and unify instructions create arguments for the subgoal procedure which is then invoked. Two of the clauses contain operations to

```
palindrome(Original) :-                             Original and Reversed
   reverse(Original, Reversed),                     are permanent variables.
   samelists(Original, Reversed).

reverse([X|L0], L) :-                               X, L1, and L are
   reverse(L0, L1),                                 permanent variables.
   concatenate(L1, [X], L).
reverse([], []).

concatenate([X|L1], L2, [X|L3]) :-
        concatenate(L1, L2, L3).
concatenate([], X, X).

samelists(Original, Duplicate) :-
        Original == Duplicate.                      equivalent built-in predicate
```

**Figure 2.5. Palindrome Prolog Operations**

```
palindrome/1:

                allocate                          allocate environment: Original
                get_variable  Y2,X1               set up argument registers
                put_variable  Y1,X2
                call  reverse/2,2                 call reverse procedure
                put_unsafe_value  Y2,X1           set up argument registers
                put_unsafe_value  Y1,X2
                deallocate                        deallocate environment
                execute  samelists/2              call samelists procedure

reverse/2:

                switch_on_term  R2c,R2a,fail      prune search space
                try_me_else  R2b                  try first clause, build choice point
R2a:            allocate                          allocate environment: X, L1, L
                get_variable  Y1,X2               retrieve argument registers
                get_list  X1                      retrieve list argument
                unify_variable  Y3
                unify_cdr  X1
                put_variable  Y2,X2               set up argument registers
                call  reverse/2,3                 recursively call reverse procedure
                put_list  X2                      set up argument registers
                unify_unsafe_value  Y3
                unify_nil
                put_unsafe_value  Y2,X1
                put_unsafe_value  Y1,X3
                deallocate                        deallocate environment
                execute  concatenate/3            call concatenate procedure
R2b:            trust_me_else  fail               try last clause, remove choice point
R2c:            get_nil  X1                       retrive argument registers
                get_nil  X2
                proceed                           return to calling procedure
```

**Figure 2.6. Palindrome WAM Code**

```
concatenate/3:

                  switch_on_term  C3c,C3a,fail         prune search space
                  try_me_else  C3b                     try first clause, build choice point
C3a:              get_list  X1                         retrive argument registers
                  unify_variable  X4
                  unify_cdr  X1
                  get_list  X3
                  unify_value  X4
                  unify_cdr  X3
                  execute  concatenate/3               call concatenate procedure
C3b:              trust_me_else  fail                  try last goal, remove choice point
C3c:              get_value  X2,X3                     retrive argument registers
                  get_nil  X1
                  proceed                              return to calling procedure

samelists/2:

                  escape  ==/2                         call built-in equivalent predicate
                  proceed                              return to calling procedure
```

**Figure 2.6. (continued)**

create and remove environments for permanent variable storage. Variables `Original` and `Reversed` are the permanent variables `Y2` and `Y1` in the palindrome predicate. Variables `X`, `L1`, and `L` are the permanent variables `Y3`, `Y2`, and `Y1` in the first clause of the reverse predicate.

The WAM code shows how choice point operations link the code for a group of clauses into a single procedure. For the reserve and concatenate procedures, which both contain two clauses, the first clause is preceded by a `try_me_else` operation to create a choice point for the procedure. The second or last clause is preceded by a `trust_me_else` operation to remove the choice point. The switch operations show how the search space of a procedure may be reduced. If the incoming first procedure argument has a constant value, the switch operation directs execution immediately to the second clause. If the incoming first argument has a list value, execution is directed immediately to the first clause. When the first procedure argument is a variable, execution passes through the switch operation and both clauses are tried.

For a fuller description of the Warren Abstract Machine, the reader is referred to Warren's original specification [39]. A description of the PLM variation of the abstract machine is given by Dobry [41]. Additional references are by David S. Warren [68] and by Hassan Ait-Kaci [69]. The first of these reports describes the abstract machine in stages building from a description of the implementation of a procedural language. The later tutorial builds the WAM by describing each design decision as a means to improve performance.

### 2.2. Special Purpose Prolog Machines

Shortly after Warren specified his abstract machine model, several groups initiated projects to design special-purpose machines that implemented the WAM instruction set and memory organization in hardware. These included Tick's pipelined Prolog processor [65, 40], the Berkeley Programmed Logic Machine (PLM) [41, 5], the Fifth Generation Computer System's Personal Sequential Inference (PSI) Machines [70, 66, 42], the Integrated Prolog Processor (IPP) [71, 72], and the Knowledge Crunching Machine (KCM) [73].

#### 2.2.1. Tick's Pipelined Prolog Processor

Working with Warren, Evan Tick designed a machine to determine the maximum Prolog performance attainable by a sequential machine for a cost comparable to high-performance Lisp machines [65, 40]. Using the register organization and memory organization of the WAM, Tick's design uses a microprogrammed organization to implement the complex WAM operations as machine instructions. Tick's objectives were a fast cycle time and the ability to issue one microinstruction per cycle when data dependencies and machine resources allow. To offset the cost of microcode interpretation, microinstructions are overlapped in a pipelined execution unit. The architecture is structured to allow multiple execution units to achieve parallel unification.

The architecture consists of a memory system, an instruction unit, an execution unit, and a microcontroller. The memory system is interleaved so that access time is not a critical factor. A cache may be placed in front of main memory.

Figure 2.7 shows the data path of the execution unit that contains three sets of resources. The first set includes WAM address pointer registers, a register file, a stack buffer, and a trail buffer. Several of the address pointer registers are counter registers to allow efficient access to sequential memory locations. The register file is a one-input, one-output array for the procedure's arguments and temporary variables. The stack buffer caches the top of the local stack. Microsequences guarantee that all references to the local stack fall within this buffer. When the current environment is below the stack buffer, it is copied to the top of the local stack. This increases the locality of stack

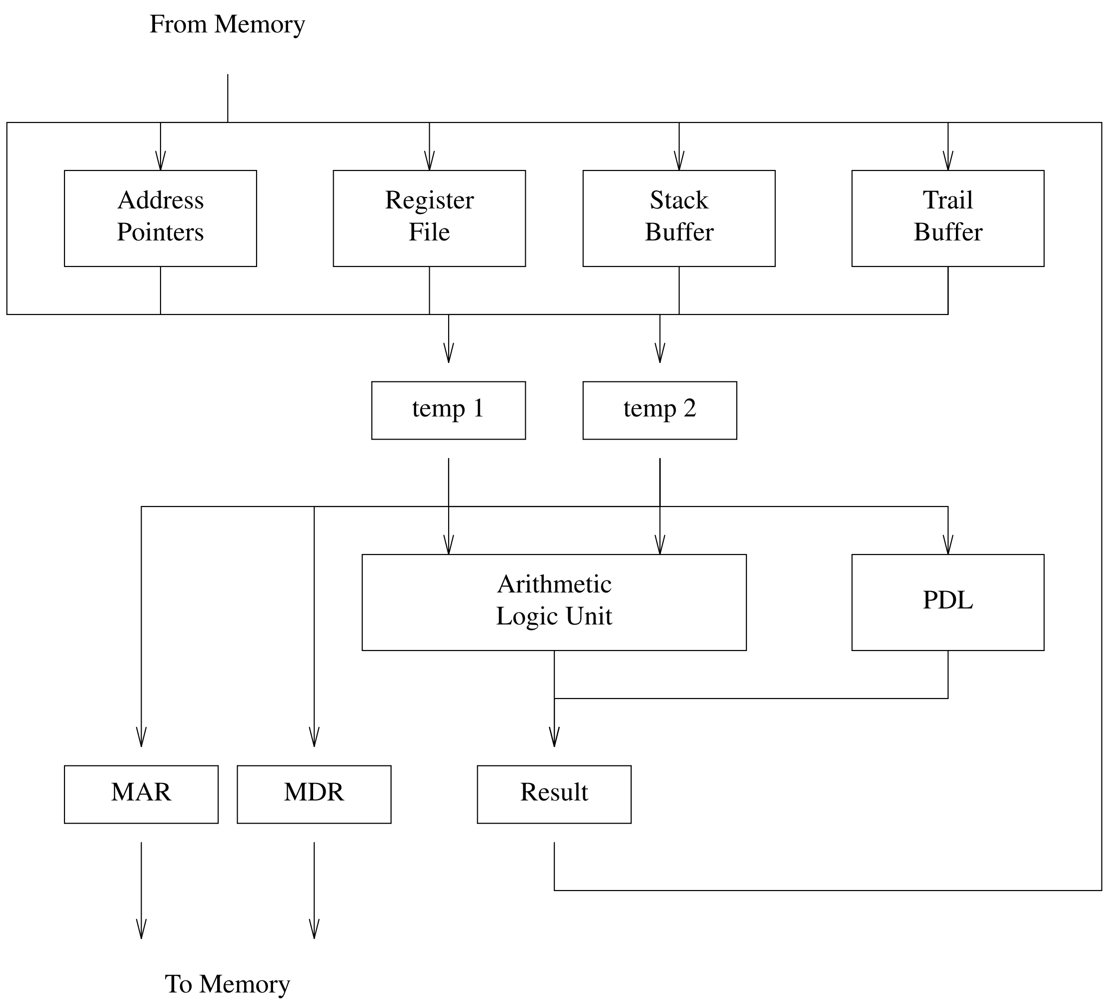


**Figure 2.7. The Data path of Tick's Prolog Processor Execution Unit**

references and creates a consistent access to the stack. The buffer uses a copy-back policy. Data consistency must be maintained between the stack buffer and all memory references. The trail buffer caches the top of the trail stack.

Two internal temporary registers form the second group of resources. The last set of resources include the arithmetic logic unit latched to a result register, an array for the push-down list, and a memory address register (MAR) and memory data register (MDR).

Tick's machine uses a three-stage pipeline. The first stage accesses the first set of resources and latches these to the temporary registers. The second stage executes the arithmetic logic unit latching results into the result register, or latches the temporary registers into the PDL or memory address and memory data registers. The last stage returns the result register to the first or second set of resources, or transfers data to or from memory.

WAM machine instructions execute using an arbitrary number of passes through the execution pipeline. The microcontroller supplies the execution unit with microinstructions for each WAM instruction. Each microinstruction specifies information for a single pipeline pass. The microstore containing the microsequences is a two-port, read-only memory permitting access to the next sequential microinstruction and a target microinstruction indicated by a branch control field. Microroutine call and return, unconditional and conditional branches, dispatch next machine instruction, and n-way branches are all supported by the microcontroller. The microcontroller is able to dispatch new instructions every cycle for nearly all microinstructions.

The instruction unit supplies as quickly as possible the starting address of a microsequence to the execution unit where it is queued. To reduce procedure call and conditional branch penalties, special attention is given to computing clause addresses. Each of the `call`, `execute`, and `proceed` operations are replaced by two instructions, a `prefetch` instruction and a `jump` instruction. The `prefetch` instruction is placed by the compiler somewhere before its corresponding `jump` instruction. It points to a `switch_on_term` instruction that dispatches to one of four clause addresses depending on the type of the first procedure argument. A small cache is included within the instruction unit indexed by procedure name to return the dispatch addresses. A `prefetch` instruction will attempt to use this cache to determine the start of the microinstruction sequence for the next procedure call together with the type of the object in the first argument register. When the next clause address of a procedure can be determined through a cached entry, the necessity of fetching the `switch_on_term` instruction is eliminated.

The instruction unit contains two interchangeable instruction buffers. These are the current instruction buffer and the future instruction buffer. Each has its own program counter. The clause address determined by a `prefetch` instruction is inserted into the program counter of the future instruction buffer. Prefetching of WAM instructions may then proceed in the future instruction buffer in competition with prefetching in the current instruction buffer. Execution of a `jump` instruction in the current instruction buffer switches the roles of the two buffers. Execution continues immediately with the WAM instructions of the next procedure.

According to Tick, the high-performance of his design and its ability to out-perform a microcode implementation on a more general-purpose machine is due to its short microinstruction cycle time, the overlapping of microinstructions, interleaved memory accesses, and the specialized hardware support for procedure calls, indexing, and unification dispatching.

### 2.2.2. The Programmed Logic Machine (PLM)

The Aquarius Project was founded in 1983 at the University of California at Berkeley by Alvin Despain and Yale Patt [74] as an attempt to achieve improvements in computer performance in applications that have substantial artificial intelligence and numerically computational components [75]. A large portion of the project involved the design of architectures and compilers for sequential Prolog execution. The first achievements were the Programmed Logic Machine (PLM) developed by Tep Dobry [41, 5] and the PLM compiler developed by Peter Van Roy [62]. The PLM is a microprogrammed engine implementing the Warren Abstract Machine. Built in 1985, using five hundred TTL chips, it was the first hardware implementation of the WAM. This work led to the VLSI-PLM [76], a single-chip WAM processor containing two hundred and fifty thousand transistors, and the Xenologic X-1 commercial Sun coprocessor.

The PLM is a coprocessor that attaches to a host central processor that provides a memory system, input and output, and assistance in implementing built-in predicates. The PLM executes modified WAM instructions that support cdr-coding, the cut operation, and built-ins predicates. PLM code space is one gigabyte of byte-addressable virtual memory. Instructions are one to six bytes in length including a one-byte opcode and zero to three operands. The PLM data space contains up to one gigabyte of virtual memory of thirty-two-bit words. Four to six bits of each word are used for tag information.

The major components of a PLM system are the host processor, the PLM Memory Interface unit (PMI), and the PLM coprocessor. The PMI is responsible for interfacing the host system to the PLM coprocessor. The PLM consists of an execution unit and a control unit. Figure 2.8 shows the configuration of these components.

The PMI coordinates activity between the PLM and the host system bus. It includes bus-protocol logic, several modules for buffering memory accesses to code space and data space, and logic for sequencing accesses to the bus. Buffering data access allows for a difference in cycle times between the PLM and its host processor. The two

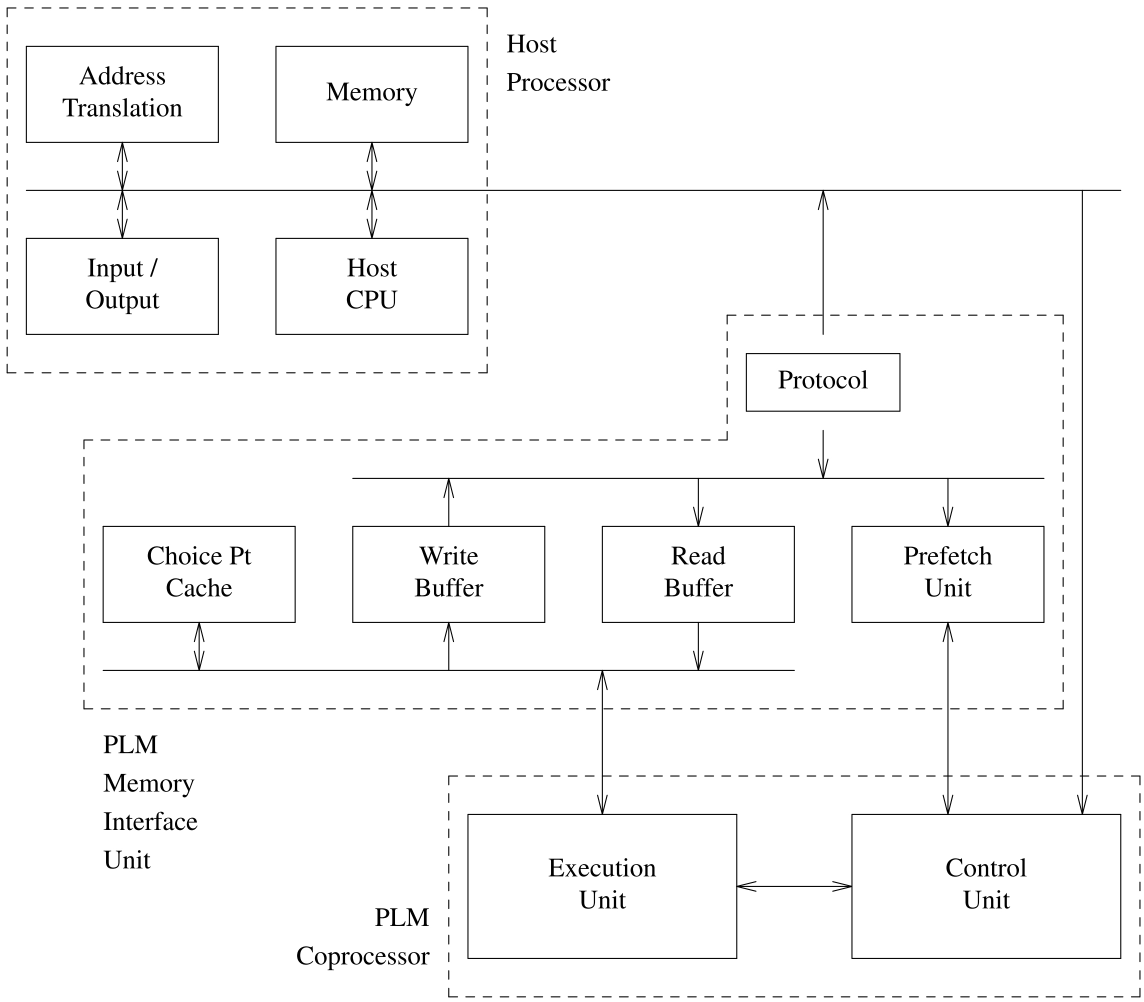


**Figure 2.8. Configuration of a PLM System**

primary data buffers are the read buffer and the write buffer. Memory write operations initiated by the PLM are queued in the write buffer waiting to be sent to memory while the PLM continues execution. PLM read requests are queued in the read buffer waiting access to the bus. The PLM waits for reads to be serviced before continuing execution. Read requests are delayed until pending writes have been serviced to ensure consistency of data.

A choice point cache holds the current choice point to speed up execution of choice point operations. A write-through policy ensures consistency between memory and this cache. When a new choice point is created, the contents of the choice point cache are overwritten and used in subsequent choice point operations. When a choice point is removed, the choice point cache is marked invalid. Refilling of the cache takes place as a side effect during a subsequent choice point access at the next fail operation. This policy eliminates premature refilling of the cache with a choice point that is overwritten before being accessed.

The PMI provides code-space buffering and partial instruction decoding in a prefetch unit. The prefetch unit contains a small instruction buffer holding up to four PLM instructions. Prefetch takes place as time permits behind data accesses as long as there is at least one instruction ready to execute. When the instruction buffer becomes empty, fetching the next instruction obtains highest priority. The prefetch unit aligns instructions and their operands before buffering. Instructions are sent to the PLM control unit through an opcode buffer and an argument buffer. The opcode buffer has single-byte width. The argument buffer holds the first operand in a thirty-two-bit word and any second or third operands packed in the next word. The PLM coprocessor will fetch the opcode and the first operand in one stage and second and third operands in a following stage.

The prefetch unit recognizes instructions that change control flow. It continues fetching instructions through unconditional branches. For conditional branches, the prefetch unit waits for the execution unit to catch up before deciding on the direction to continue prefetching. Backtracking is an additional change of control flow. It causes the prefetch unit to flush its instruction buffer and begin fetching at the backtrack point in parallel with the restoration of state information from the choice point contents.

The PLM processor consists of an execution unit and a control unit. The control unit includes a microsequencer and a kiloword control store containing 144-bit-wide horizontal microinstructions. The control unit coordinates the PMI and the PLM execution unit. It interfaces directly with the host so that the host may start and stop control of the PLM and inspect its state.

The control unit dispatches signals to control the data paths of the execution unit. It calculates the next microinstruction address and fetches the microinstruction from the control store. These operations are executed in parallel.

The next instruction address is derived in several stages involving various decisions. The source of the next instruction may come from machine-instruction decode logic, fundamental-operation select logic, a microreturn register, or the microaddress seed of the current microinstruction. The instruction decode logic determines an appropriate microcode entry point for the execution of a new PLM instruction using the instruction's opcode and some status information. The fundamental-operation select logic is used to branch to a microcode entry point other than the start of a new PLM instruction. This occurs most frequently in executing fundamental operations. The microreturn register is used to implement microcall and microreturn operations for the dereference and decdr operations. The microaddress seed specifies a branching address for the microcode sequences. It may be used to realize up to a four-way branch by dividing the control store into four pages, prepending two page bits to the microaddress seed, and then branching to the address given by the seed in the selected page. To allow all decisions to be made and time for an access to the control store to be accomplished in a single microcycle, several features had to be added to microcode and hardware including delayed branching and shadowing of the tag bits for all the argument registers.

The execution unit consists of register files, an arithmetic logic unit, internal temporary registers, memory-interface registers, and a push-down list array much like the execution unit of Tick's machine. These are interconnected through six major busses in three bus pairs. The PLM does not contain a stack buffer. Execution of PLM instructions is analogous to the three-stage pipeline of Tick's processor. Microcode controlling the execution unit is written to utilize the stages in parallel as much as possible within the microsequence for each PLM instruction. A typical PLM instruction may involve one first-stage operation, several second-stage operations, and a final put-away operation. Overlap between PLM instructions is not supported.

The PLM implements built-in predicates in three ways. Several built-in predicates are implemented using the existing instruction set. Some built-in predicates are implemented through microsequences using the execution unit data path. Most built-in predicates are implemented by use of an escape mechanism. A PLM escape instruction causes the start of the escape microsequence. In the sequence, argument registers are written to memory. The microsequencer signals the host and enters a wait state suspending microprocessing. The host executes the built-in operation. Upon completion, the host signals the PLM and the microsequencer resumes processing the escape microsequence. The argument registers are reloaded to obtain the results of the built-in predicate routine. The microinstruction sequence of the next PLM instruction is initiated.

As seen by the PLM developers, the advantages of the PLM are the simplification of the compilation task, a tagged architecture, hardware hashing for efficient filtering of clauses, support for the instruction set through specialized registers for the stacks areas, the containment of the PDL within the processor, and parallelism and cycle reduction in the microcode.

### 2.2.3. The Personal Sequential Inference (PSI) Machine

The Japanese Fifth Generation Computer Systems (FGCS) Project was announced in 1981 to develop a new generation of computers suited to knowledge information processing and centered around logic programming [77, 3, 4]. The project developed a hierarchy of languages starting with a logic programming machine language and building up to a knowledge representation language [2, 78] with new hardware architectures to support these languages.

The Kernel Language Version 0 (KL0) [79], designed as a sequential machine language, is a low-level derivative of Prolog with extensions for built-in predicates, modularization, data-type checking, interfacing to relational databases, and operating system support. The Personal Sequential Inference Machine I (PSI-I) [70, 80, 66, 81, 82] is a single-user, self-contained, multiprocess machine with KL0 as its target language. The PSI-I is built from two thousand high-speed Schottky TTL MSI circuits and MOS RAM integrated circuits on twelve printed circuit boards running at a two-hundred-nanosecond clock speed. It includes eighty megabytes of main memory, an eight-kilobyte, two-set, set-associative, mixed instruction and data cache, and the ability to support sixty-four concurrent processes. The PSI-I's model of execution is Warren's original model as developed in his thesis [38]. KL0 is translated into an internal table type of machine code by a simplistic assembler. Execution of the assembled code requires a complicated microprogrammed interpreter. The operating system of the PSI-I is the Sequential Inference Machine Programming and Operating System (SIMPOS) written in KL0 [2]

The PSI-II is the successor to the PSI-I. Its design objectives were to reduce the amount and size of hardware and to improve performance in comparison to its predecessor, and to serve as an element of a parallel inference machine. The PSI-II is a desk-side logic programming workstation. It is implemented in CMOS gate array LSI circuitry. Four standard chips are used in the design of the arithmetic logic unit and multiport register files. Nine custom chips complete the central processor unit design. The total logic is packed onto three printed circuit boards. Execution is at a clock speed of 177 nanoseconds. Much of the performance improvement of the PSI-II results from a change to the Warren Abstract Machine execution model. While the shift from the PSI-I to the PSI-II increases complexity through the need of a KL0 to WAM translator, the instructions of the WAM may be executed at higher speed than the machine code of the PSI-I [83]. Improvements in speed are also due to specialized hardware components for multiple-stack management and tagged data manipulation.

The majority of the PSI-II's attributes are inherited from its predecessor including multiple concurrent processes, microprogramming, and the design of its memory organization, address translation mechanism, and cache. The choice to model the Warren Abstract Machine is reflected in a new instruction set, data processing unit, and sequence controller.

The logical address space of the PSI-II is divided into eight areas identified by the three highest bits of a thirty-two-bit address. Three of these areas correspond to the global stack, local stack, and trail stack of the WAM. These areas are local to each process. Two additional areas are the code space and the system space that includes process control blocks, interrupt vectors, and memory management tables. These two areas are shared by all processes.

The word size of the PSI-II is forty bits for both instructions and data. PSI-II instructions are one or more forty-bit words. The two highest bits of the first word of an instruction serve as a classification field. The first instruction class contains instructions that execute the WAM operations. The second class of instructions are built-in predicate instructions. Built-in predicates of KL0 are directly executed by microcode. In addition to the built-in predicates provided by Prolog, KL0 contains built-ins for execution control and operating system support.

Each instruction class contains 256 instructions. The twenty-four-bit operand field of the first instruction word may be used to specify one, two, or three operands. Second and consequent words of an instruction may be used to represent forty-bit atomic operands containing a tag and value field. Two bits from the tag field are used in garbage collection. PSI data words are separated into an eight-bit tag field and a thirty-two-bit value field. Data types include the basic WAM types, floating point numbers, and strings.

The components of a PSI-II system are a main memory, an address translator, and cache memory, a data processing unit, a sequence controller, and an input/output bus interface.

The PSI-II has a sixty-four-megaword memory size allowing the implementation of memory consuming applications without the need of virtual storage. Each stack area may grow or shrink freely without any collision nor copying as long as the total of their sizes does not exceed memory size. This is achieved by an address translation mechanism similar to segmentation in virtual memory systems. The scheme is shown in figure 2.9. A PSI-II process has four gigawords of logical address space represented by a thirty-two-bit address and translated into a twenty-six-bit physical address by a two-level table look-up. The top three bits of the logical address determine a memory

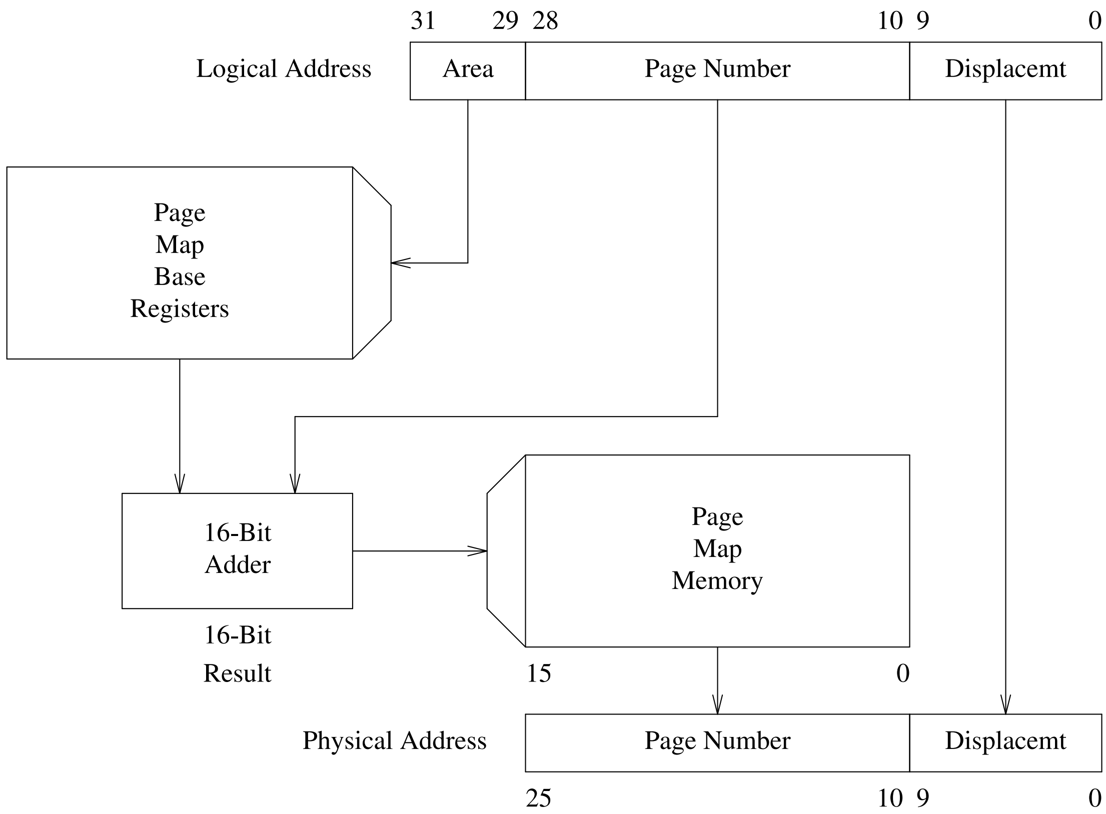


**Figure 2.9. PSI-II Address Translation Scheme**

area (one of the WAM stacks) of the address space. The middle nineteen bits determine an offset into a page-map for the area. The last ten bits are a displacement within a physical page. The area bits are used to select one of eight page-map base registers particular to each process. A page-map base register contains the base address of a set of entries in the page-map memory, a high-speed memory area shared among all processes. The page-map base address and the logical page number are added to reference the page-map memory to obtain a physical page number. The physical page number and displacement field are combined to obtain a physical address. This address translation scheme is performed in a single microcycle by the microprogrammed memory manager. Traps to the SIMPOS operating system handle the allocation of pages to a stack area from a free page list, the deallocation of unused pages in a stack area, and the distribution (and possibly relocation) of page tables within the page-map memory.

The PSI-II cache provides high-speed access to the code and data areas. The cache size of the PSI-II is forty bits by four kilowords. The cache uses a write-back policy. An optimized instruction exists for the growth of stacks. When a write operation misses the cache, normal operation loads a memory block from main memory to validate the other words of its block in the cache. When pushing a value onto a stack, the words above the top of the stack are not meaningful and can be disregarded. When pushing to the first word of a cache memory block, there is no need to load a block from memory even if it misses the cache.

The layout of the data processing element is shown in figure 2.10. An arithmetic logic unit, a tag comparison unit, a register file, and several additional registers are connected with two forty-bit-wide source data buses and a forty-bit-wide destination bus. The lines of the tag and value fields of data on these buses separate into subbusses. Lines of the source and destination value fields lead to and from the arithmetic logic unit. Lines of the source tag fields lead to a tag comparison unit. The arithmetic logic unit performs addition, subtraction, logical operations, multiple-bit shift, and bit-field manipulation for value fields. The tag comparison unit compares tag fields of operands to speed up the manipulation of the tag bits.

The register file contains sixty-four forty-bit words. The first half of the file is used for arguments and temporary variables. The second half is used for WAM address pointers. The program counter, the top of global stack register, and a register used to traverse structures and lists are specialized. These last two registers are implemented as counters to assist in sequential memory access.

The instruction register (IR), the instruction buffer register (IBR), and the instruction fetch registers (IFR) are used for prefetching machine instructions. The instruction to be executed is held in the instruction register. The instruction buffer register holds the next instruction to execute. The instruction address register holds the address of the instruction following the next instruction. The last microinstruction of a KL0 microinstruction sequence will dispatch to the beginning of the next microsequence by transferring the contents of the instruction buffer register to the instruction register, and fetching the instruction pointed by the instruction address register into the instruction buffer register.

The PSI-II sequence controller contains a sixteen-kiloword, fifty-three-bit writable control store to obtain microprogrammed execution of KL0 programs. Most branch operations within the control store are two-way branches. The sequence controller uses more sophisticated branch instructions as well. The microbranch operations

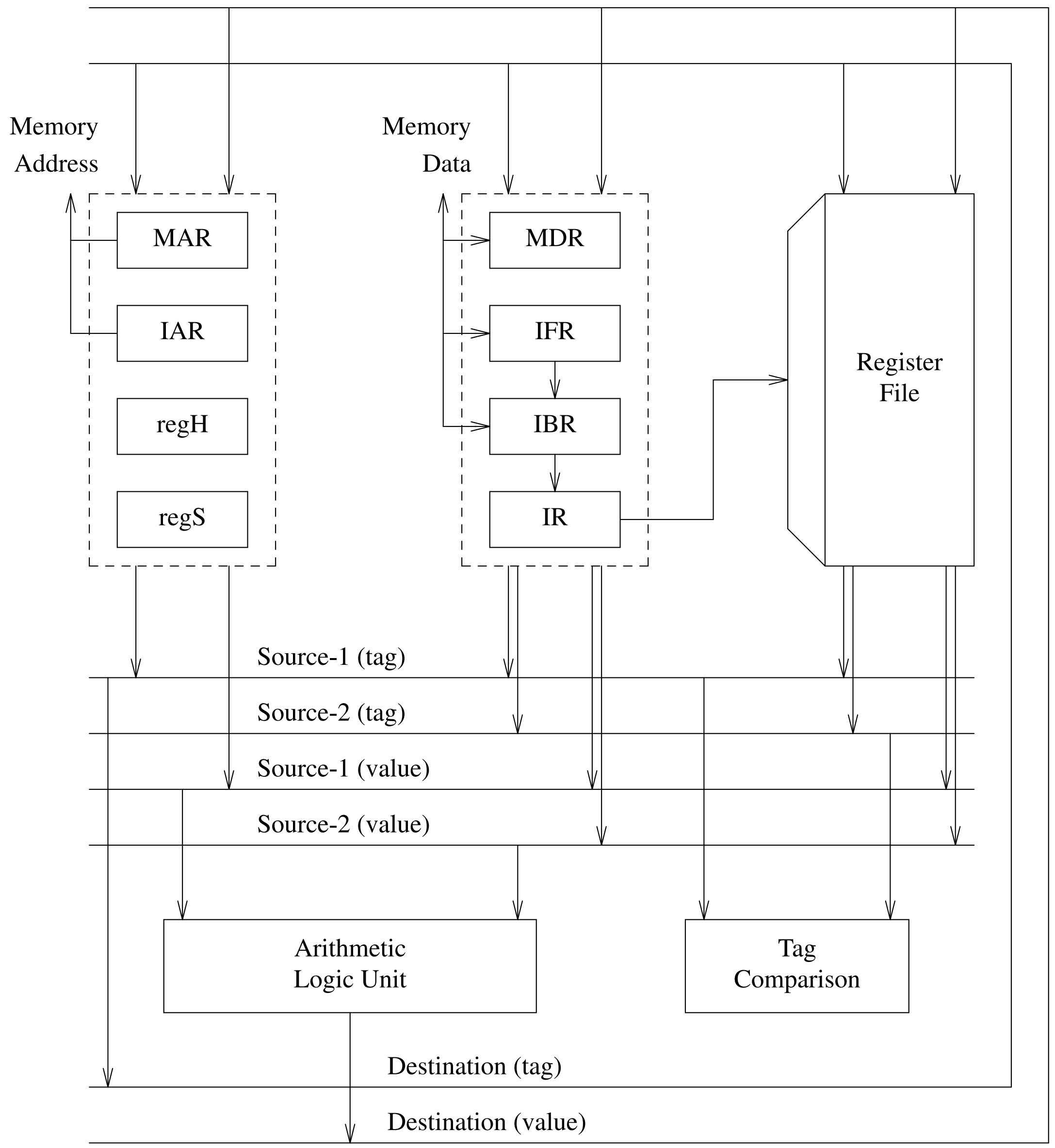


**Figure 2.10. PSI-II Data Processing Element**

include a direct branch with relative or absolute addressing, a microprogram call and return to a depth of sixteen microroutines, an indirect branch using an indirect jump register, instruction code dispatching using a one-thousand-entry RAM table, and data type dispatching using a two-thousand-entry RAM table. The instruction code dispatcher is used to select the start of the sequence of microinstructions for each KL0 instruction. The data type dispatcher is used to select a portion of this sequence based on the operand types of the instruction. The table for data type dispatching is partitioned into thirty-two blocks of sixty-four entries. Each corresponding entry corresponds to a data type and contains an offset value. The content of the tag field of the memory data register (MDR) or instruction fetch register (IFR) is concatenated with the block number specified in the microinstruction to generate a table entry address. The offset value within the table entry is `OR`'ed with a base address specified in the microinstruction to obtain the next microaddress. This data type branch is useful in checking tag conditions, including comparison of the tag of the first source bus with an immediate value in a microinstruction, and in the comparison of the tag part of the two source buses to determine whether two data have the same type. It is possible to combine a tag equality condition with an arithmetic logic unit condition in a single microinstruction.

The strengths of the PSI-II come from enhancements to Prolog offering a greater number of built-in operations including low-level system functions, a large main memory, a fast and powerful address translation mechanism, inclusion of a tag comparison unit, and sophisticated microbranch operations.

### 2.3. Prolog RISC Implementations

As computer architects exploring efficient implementations for numerical computation began to debate how closely an instruction set should match the semantics of a particular high-level language and the performance advantages of a reduced-instruction-set-computer (RISC) architecture became more apparent, the attraction of designing specialized Prolog processors with complex instruction sets diminished. Attention turned to the design and tuning of RISC instruction sets for Prolog steered by the work of a group at the University of California at Berkeley that compared the performance of Prolog running on the PLM to Prolog executing on SPUR (Symbolic Programming Using RISC) [43, 84] and a Prolog RISC architecture proposed by the LOW RISC group at Arizona State University [85, 86, 6].

#### 2.3.1. Prolog on the Symbolic Programming Using RISC (SPUR) Machine

The proximity of work on RISC architectures at Berkeley, specifically the design of the SPUR (Symbolic Processing Using RISCs) processor, to Berkeley's PLM project led to a study by Borriello, Cherenson, Danzig, and Nelson comparing the expected performance of Prolog running on the PLM to Prolog executed on SPUR [43].

SPUR is a high-performance, multiprocessor, personal workstation designed for general-purpose processing with support for Lisp and floating point computations [84]. SPUR extends the work of the RISC [44] and SOAR [87] architectures as a reduced-instruction-set computer with extensions for tagged data, a large mixed instruction and data cache, and a tightly-coupled coprocessor interface.

The architecture uses forty-bit internal words that contain an eight-bit tag. External memory is thirty-two bits wide where tagged data is aligned in two words (sixty-four bits) in which the first word contains the data or pointer and the second contains the tag. Busses within the processor and cache are forty-bits wide so that after data enters the cache no further overhead is incurred in transporting tagged information.

SPUR uses thirty-two-bit wide instructions including register-to-register data transfers and arithmetic and logical operations, memory-to-register load instructions, register-to-memory store instructions, and instructions for comparison and branching. The majority of instructions are register-to-register involving forty-bit tagged quantities. Several instructions call exception handlers when detecting explicit tag conditions. For example, an add operation is included that adds the thirty-two-bit data fields of two operands as integer values. In parallel, the operands' tags are checked to verify that the operands are indeed integers. If both operands are not integers, the operation is redirected to a software routine. Other instructions operate only on the tag field treated as a single entity. Load and store variations are included for tagged and untagged data. Comparison and branch instructions alter control flow by examining the value of a CPU register, a coprocessor register, or a tag field. Instructions contain fixed positions for opcode and register fields eliminating the need for alignment or pre-decoding. Instructions execute in a single, one-hundred-nanosecond cycle in a four-stage pipeline with the exception of two-cycle stores. The effects of loads and branches are delayed one cycle.

Memory is organized as a thirty-eight-bit virtual-address global space with a thirty-two-bit virtual-address process space. The memory hierarchy includes an on-chip instruction buffer of 128 words and an off-chip mixed instruction and data cache of 128 bytes.

SPUR includes 138 general-purpose registers. The registers are partitioned into eight overlapped windows. Ten global registers are accessible in all windows. In each individual window, six input registers overlap with the previous window, ten registers are local, and six output registers overlap with the next window. During a procedure call the current window shifts to allow input parameters and output results to be passed between procedures.

In the Berkeley study, the translation from Prolog to SPUR code is achieved by a simple macro-expansion of PLM instructions after compilation by the PLM compiler. This technique establishes a lower bound on the expected performance of Prolog on SPUR. Macros closely follow the semantics of the PLM simulator with exceptions of the use of SPUR's tags to simulate PLM tags and the use of register windows for choice points.

Variables of environments and the global stack are not conveniently assigned to registers since registers lack addresses disallowing them from being bound to other variables; however, the size of SPUR's register windows and their overlapping and stack-like characteristics fit the needs of the choice point buffer. Table 2.2 shows the arrangement of a choice point within a register window.

Global registers are used as argument registers for arguments passed to the current predicate and for those passed in the next predicate call. The output registers store the current PLM address pointers. When a new choice point needs to be created, the current argument registers are copied from the global registers to the local registers and the register windows are shifted to save the previous choice point. By storing choice point data in register windows and not in a stack frame in memory, the number of stores to memory required for choice point operations are reduced. Backtracking is accomplished by shifting the register windows and transferring register values eliminating a large number of fetches from memory. The overlap of register windows allows the values of the previous choice point to be accessible within the current choice point.

| Type | Register | Use |
|---|---|---|
| Globals | 0 | hardwired constant zero |
| | 1-8 | current predicate arguments |
| | 9 | pointer to constant table |
| Inputs | 10-15 | choice point address pointers |
| Locals | 16-17 | linkage and temporaries |
| | 18-25 | choice point predicate arguments when window becomes choice point |
| Outputs | 26-31 | current address pointers |

**Table 2.2. Register Usage of SPUR PLM Implementation**

Register windows are less effective in assisting with recursive unification since their size is significantly larger than the three arguments used in each unification call. Instead, a procedure call mechanism is employed with a memory stack to avoid a register window shift. A procedure call mechanism is also used to assist in the implementation of built-in predicates. Built-in predicate arguments and a return address are transferred to temporary registers. Execution continues with a procedure for the built-in predicate that accesses and/or updates the values of these registers that are copied back to their original locations after the procedure returns.

The PLM versus SPUR results are based upon a number of simulations using the PLM and SPUR simulators with fourteen standard benchmarks [5]. Using techniques, assumptions, and simplifications similar to those of Dobry, several low-level statistics (that is, instruction counts, memory references, and cache misses) are recorded and used to arrive at first-order approximations in the reported results. The SPUR implementation is found to have an average static code size of seventeen times larger than PLM code size. SPUR executes an average of sixteen instructions for each PLM instruction.

By translating instructions executed into cycles executed, and equating both machines with the same cycle times, and assuming an ideal memory system that supplies instructions and data immediately as needed to the processor, SPUR achieves the performance of forty percent of the PLM. Accounting for a memory system in which each machine is configured with its own instruction buffer design and with SPUR's mixed cache increases the execution times for each simulation to account for cache misses, but results in a decrease in the relative performance of SPUR to twenty-nine percent of the PLM. Relative performance degradation is due mostly to the PLM instruction buffer's ability to prefetch virtually all instructions, (as PLM code lacks many conditional branches and allows a large amount of computation to overlap a conditional branch when one does occur), and the SPUR instruction cache's limitation of prefetching a single instruction at a time. It is expected that widening the prefetch to two words and further doubling the size of SPUR's instruction buffer would improve its miss ratio by forty percent. The large code size of Prolog programs running on SPUR makes these changes highly desirable.

Several optimizations were suggested and estimated. A global constant propagation optimizer was estimated to improve SPUR performance by thirty percent. Fifty to one hundred percent speed-up was shown possible with hand-optimizations of a few benchmarks. Adding an orthogonal tag check to the SPUR instruction set is expected to increase performance by fifteen percent.

A more substantial enhancement was the suggestion of designing a tightly-coupled coprocessor for Prolog on the processor/cache side of the SPUR system bus. The group identified the basic operations performed by the Prolog and determined which could be performed efficiently with standard SPUR instructions and which would benefit from the use of coprocessor hardware. The tightly-coupled coprocessor would be similar in design to the PLM with similar instructions. It would greatly reduce the static code size and was estimated to overall achieve a ten percent performance improvement over the PLM due to an exploitation of macroinstruction overlap.

The major contribution of the Berkeley group was questioning whether the complexity and cost involved in building a language-specific, loosely-coupled coprocessor is justified when a general-purpose processor like SPUR can provide competitive performance. By simple macro-expansion of a Prolog-specific instruction set to a RISC instruction set, the group reported that Prolog programs can execute at twenty-nine percent the performance of the Prolog-specific machine. With minor architectural support and compiler optimizations, Prolog on SPUR runs at least fifty percent of Prolog on the PLM.

### 2.3.2. The Logic Programming Windowed (LOW) RISC Machine

Because a conventional RISC architecture has less data path parallelism than a dedicated WAM machine, and because translating Prolog to RISC code introduces many conditional branches and procedure calls that do not exist in equivalent WAM code, and because RISC code is significantly larger than equivalent WAM code, designing a RISC Prolog architecture was initially considered unproductive. By concluding that only half the performance of the PLM could be achieved on the SPUR and discussing the design of a special-purpose coprocessor to attach to the SPUR, , the SPUR study added to the hesitation to design a Prolog machine through a RISC philosophy.

Jonathon Mills proposed the Logic Programming Windowed (LOW) RISC machine arguing that Prolog performance on SPUR suffered because SPUR was designed to support Lisp and not Prolog. Performance improvements greater than those obtained by the SPUR group are possible with the LOW RISC due to an increased number of data paths inside the LOW RISC allowing tag and value operations to be performed in parallel, an instruction set evolved from the Warren Abstract Machine, and hardware support for operations common to Prolog such as tag checking and branching on tag fields. Hardware support for shallow backtracking, stack manipulation, trailing, and partial unification could further increase performance.

The simple, unoptimized macro-expansion used in the SPUR study leads to a slow implementation. The LOW RISC implementation views the WAM as an abstract compiling paradigm in which RISC code is deleted, modified, or moved while maintaining the semantics of the abstract machine. Optimized code does not correspond to a sequence of WAM operations but instead to a sequence of pieces of WAM operations corresponding to those parts of a WAM operation executed in the specific context of a program. In certain contexts, by looking within a WAM operation, suboperations such as trailing and dereferencing can be identified as unnecessary and be omitted from the RISC code. Strength reduction can be used to decrease the number of cases that must be examined when the types of either or both of two arguments to be unified is known at compile time.

By estimating the cycle times and code size of LOW RISC routines and comparing these values with the cycle times and code size of PLM instructions, Mills found LOW RISC code size to be seven times larger than PLM code size and claimed that the LOW RISC machine could achieve twice the performance of the PLM.

The LOW RISC computer is a thirty-two-bit, register-oriented von Neumann machine with fixed-width, single-cycle, pipelined instructions that complete every half-cycle once the pipeline is filled. A half-gigaword memory is organized where each word contains a three-bit tag field and a twenty-nine-bit value field. The original LOW RISC design resembles a minimal Prolog machine and depends heavily on a host processor for arithmetic and operating system support. The final LOW RISC design looks more like a general-purpose processor although it is less suited for tasks other than symbolic processing. Real arithmetic and other built-in predicate operations must be handled by an attached processor.

The original LOW RISC includes only seven instructions. Memory references are achieved through delayed load and store instructions. These instructions set tag status flags to indicate the type of data fetched or stored. Add and subtract instructions perform tag and value manipulation in parallel. Both fields of two operand registers are independently added or subtracted and the result is stored in a destination register. The add instruction may also operate as a logical instruction. Control bits allow the add instruction to zero one operand's value field and the other operand's tag field to combine together a tag and value field into a single result. In addition and subtraction, both tag and value status flags are set to reflect the resulting value. Unconditional branching may be achieved through an explicit branch instruction, by loading a value into the program counter, or by performing an arithmetic operation and storing the result in the program counter. A conditional branch chooses whether or not to transfer control dependent upon the tag and value status flags. A switch instruction provides transfer of control using a single register's tag field to select one of four offsets. A hook instruction serves to interface to the attached processor.

A fair amount of simulation [86] prompted the group to enhance its architecture resulting in a doubling in the number of instructions. Additional integer arithmetic and logical instructions eliminate nearly all calls to the attached processor. Logical operations facilitate other methods of indexing clauses. The switch instruction is refined to a three-way branch in which an offset is shifted one bit leftward before being added to the program counter. Other enhancements include a method to provide hardware support for garbage collection through a dedicated bit that is used either as a mark or reverse bit, a data structure overflow warning method provided by hardware to check for global stack, local stack, and trail overflow, and a dereference instruction providing an addressing mode to follow chained references.

The LOW RISC design includes the LOW RISC processor, an instruction cache, a data cache, and an attached processor, shown in figure 2.11. The LOW RISC processor shown in figure 2.12. is a Harvard bus machine. Three pipelined arithmetic logic units operate in parallel each performing a unique function. The tag ALU selects the destination tag. The value ALU is the only unit that performs thirty-two-bit addition and subtraction. It determines the destination value, calculates the address of operands, and performs register arithmetic with the program counter. The address ALU increments the program counter and calculates branch addresses from the immediate offsets in branch and switch instructions. The output of the value ALU or the result of a load overrides the address ALU output if the destination is the program counter.

The LOW RISC processor organizes its registers into four groups: general-purpose registers, control registers, global registers, and a status register. The 115 general-purpose registers are organized as a variable-sized, windowed-register file in which at most nineteen consecutive registers are accessible at any one time. The window pointer control register is used as a base register into the register file to select the accessible registers. General-purpose registers are used to store the choice point stack and to assist unification in handling nested lists and structures. Five global registers are always accessible. Five of seven control registers are used to hold WAM address pointers.

The instruction cache and compiler work to take advantage of locality of reference exhibited during head unification and tail recursion. The compiler produces blocks of three to ten instructions of in-line code for the head of a

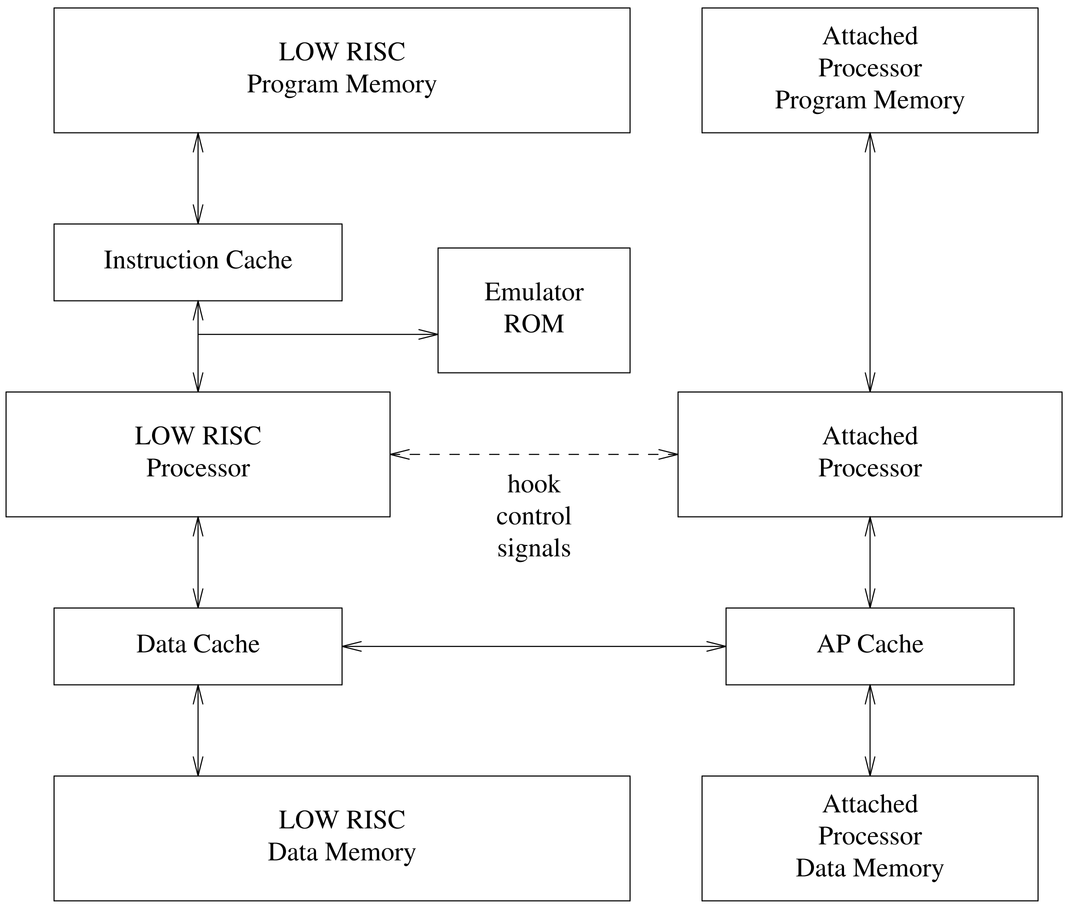


**Figure 2.11. LOW RISC System Organization**

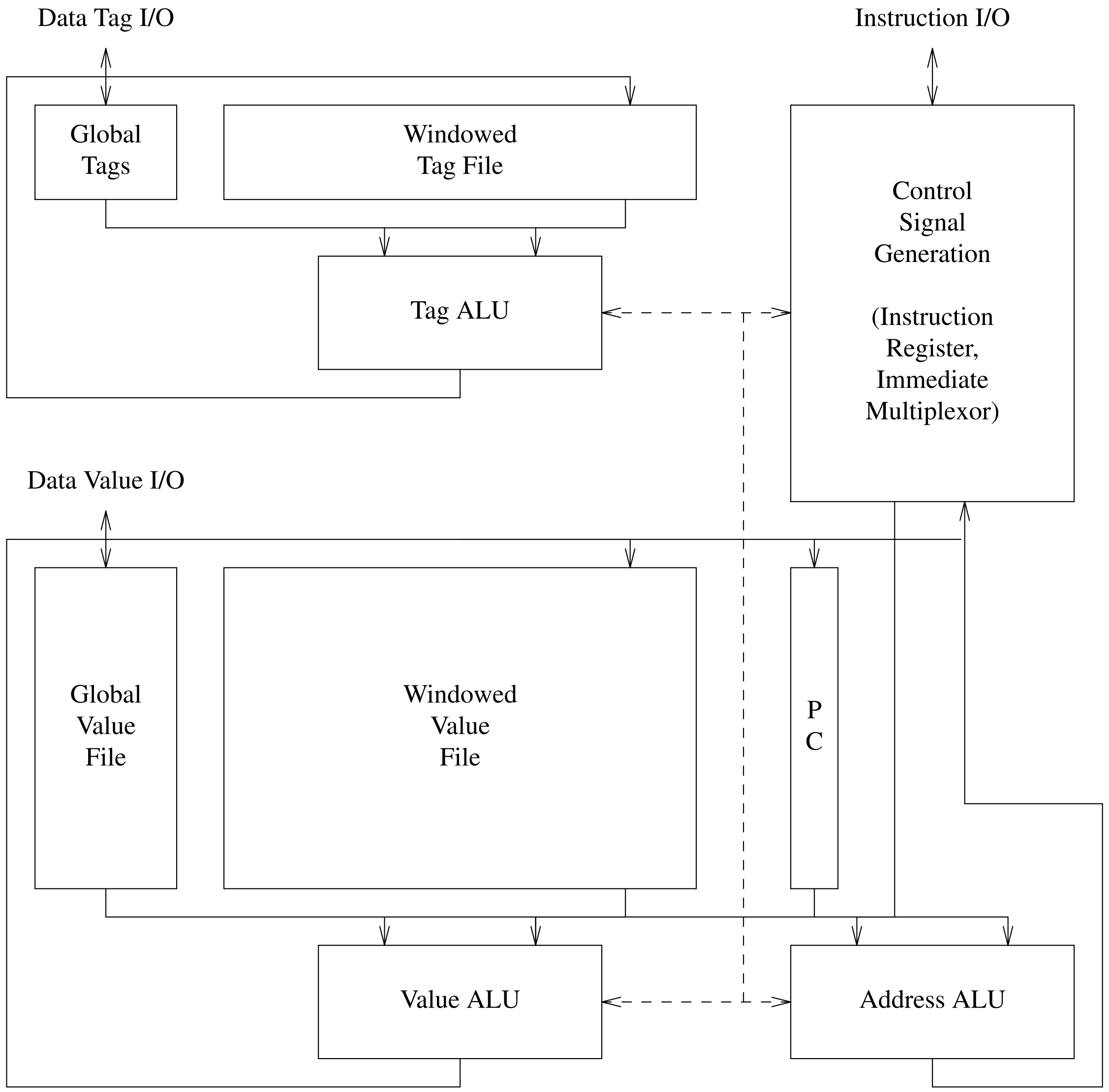


**Figure 2.12. LOW RISC Processor Organization**

clause linked by forward references. The instruction cache prefetches a four-word block to allow the LOW RISC to execute the head of a Prolog clause with few cache misses. (Cache misses are more frequent when a goal is called and at the termination of a clause.) The data cache supports memory references into the global stack, the trail, and the local stack. It also functions as a communication path to the attached processor. Both the instruction cache and data cache are small compared to main memory so that they may be constructed of memory fast enough to allow the LOW RISC to run at full speed as often as the cache hit rate allows. The attached processor performs system functions such as floating point arithmetic and other built-in predicate functions that cannot be performed by the LOW RISC processor.

According to Mills, the LOW RISC executes Prolog quickly due to its simple instruction set tailored to Prolog that allows WAM operations and Prolog goals and procedures to be coded efficiently. LOW RISC allows optimizations that are not possible with a WAM instruction set and allows a Prolog implementation to keep pace with evolving compiler technology. Enhancements can be added as improvements to the compiler rather than as hardware or microcode changes.

### 2.4. Global Prolog Compiler Optimizations

As the development of efficient Prolog architectures proceeded, compiler researchers made equal progress towards efficient Prolog compilation by exploiting global optimization techniques through abstract interpretation [88-95]. Abstract interpretation is a technique in which a program is reinterpreted in simpler, more fundamental ways so that attributes of the program can be more easily observed and understood. A Prolog program is given new interpretations consistent with its declarative and procedural semantics. The program is viewed as computing in a domain where less information about the data objects is considered. Results (characteristics of the program) are more feasibly computed in the abstract domain and reflect properties in the program's standard operation. This method has been studied extensively and developed into a practical compilation tool [96, 97]. Representative of the abstract interpretation and global optimization work are efforts by Saumya Debry and David S. Warren [52, 51] and C. S. Mellish [49, 50] on detecting determinism and automatic mode determination and Peter Van Roy's development of the Aquarius compiler [54, 55].

#### 2.4.1. Mellish, Debray, and Warren

In logic, it is frequently the case that there are alternative ways to show that some theorem follows from a set of axioms. This leads to the notion of nondeterminism in logic programming. Depth-first backtracking, which implements nondeterminism in Prolog, is known to be inefficient. Many projects have avoided using Prolog's search mechanism in favor of more intelligent strategies expressed in the form of Prolog programs that are largely deterministic. Many computation problems are not most naturally conceived in terms of nondeterministic specifications. Although nondeterminism is often used as a local control structure, it is infrequently used throughout a Prolog program.

In logic, axioms involving a predicate can be used to prove ground sentences as well as existentially quantified sentences. This corresponds to the logic programming concept of a multidirectional procedure in which there is no designation of input and output arguments in a procedure. Unknown values are computed from the known values in whichever way is required. It is difficult to write multidirectional programs in Prolog. Prolog's order of clause selection and depth-first search make it easy for programs to enter infinite loops when procedures are used in unexpected directions. Many of Prolog's built-in predicates are directional. This directionality is inherited by procedures that call them directly or indirectly. Although there are well-known examples of multidirectional Prolog programs, most Prolog procedures are built largely with a specific direction in mind.

Because many Prolog programs do not make full use of the flexibility of the language, the control structure of many parts of Prolog programs are similar to those of conventional programs. Global analysis and detection of program segments that use a restricted set of Prolog's facilities allow Prolog compilers to produce code of comparable efficiency to conventional languages for those particular segments. Mellish, Debray, and Warren have been able to automatically detect parts of programs that are restricted in terms of determinacy and procedure argument directionality [52, 51, 49, 50].

A deterministic predicate is a Prolog procedure whose definition and use determine that it is never possible for a goal involving that predicate to return more than one possible solution. Deterministic procedures will never backtrack and find alternative solutions. Knowing that a procedure is deterministic may make it possible to avoid the creation of choice points, to allow early reclamation of space on the local stack, and to avoid unnecessary search. The compiler may build sophisticated indices or transform a program based on "mode" information and analysis of the mutual exclusion of clauses so that Prolog's indexing schemes eliminate the need to create choice points.

The concept of "modes" was introduced by Warren as a way of describing the ways in which a predicate is used in a Prolog program [38]. A simplistic view of modes is that a particular argument within a procedure may always have an instantiated term passed to it, or may always have an uninstantiated term passed to it, or both situations may be possible. When it is known at compile time that a particular predicate argument will always be instantiated or will always be uninstantiated, the unification code that is produced for the argument may test for fewer possibilities than if this information was not known. This often leads to faster and more compact code. Mode determination allows optimizations in the dereferencing of terms. Because the unification of one predicate argument can cause other arguments to become further instantiated, it is inadequate to dereference all predicate arguments in advance. Each argument must be dereferenced each time it is subjected to a unification test in a clause. Repetitive dereferencing can be avoided when mode analysis can determine the point at which a variable is instantiated. Mode information is also useful in further analysis of the program. It can assist in the detection of deterministic predicates.

### 2.4.2. The Aquarius Compiler

Peter Van Roy's extensive work on Prolog compilation led to the development of the Aquarius compiler [54, 55] for the Berkeley Abstract Machine (BAM) processor [7]. Van Roy includes significant dataflow analysis and global optimizations in his compilation scheme to the point where he feels it is less useful to think in terms of the Warren Abstract Machine. Instead, Van Roy uses knowledge of possible compiler optimizations applied to the semantics of Prolog to decompose Prolog's general operations into basic components. These components form the Berkeley Abstract Machine (BAM) execution model consisting of instructions and addressing modes required to compile Prolog operations into efficient code through extensive optimizations and compact encoding. The BAM model uses data structures similar to those of the WAM. (One noted difference is that BAM variables are restricted to the global stack.) It includes simple tagged data-transfer operations, more complex operations that assist environment and choice point handling, dereferencing, trailing, and general-unification, and non-executable information that assists the optimization of the target machine assembly language. The compiler transforms Prolog into an internal representation called kernel Prolog, performs dataflow analysis on the kernel Prolog, and compiles the resulting annotated kernel Prolog into BAM code. BAM code is macro-expanded into assembly code and processed by a peephole optimizer and instruction reordering stage to produce efficient code for the BAM processor.

The compiler's dataflow analysis derives procedure argument type information including ground, nonvariable, dereferenced, and uninitialized variable modes. The compiler generalizes the first argument selection of the WAM by creating a decision graph that represents the possible types that are unifiable for each of a procedure's arguments. Derived type information is used to simplify the graph. The graph is exploited to produce special-case unification code and minimal sequences of clauses to be considered during a procedure's execution. By making intelligent use of procedure argument information, the compiler replaces the creation of choice points and selection of clauses by unification and failure with deterministic testing and branching in an effort to achieve an efficiency comparable to that of simple test and indexed jumps in conventional languages. The compiler keeps track of which variables are dereferenced and generates explicit dereferencing code only when necessary.

The notion of “uninitialized” variables are used to overcome a WAM inefficiency in which variables are created (initialized) and unified soon afterwards. Unifying an initialized variable is expensive because it includes dereferencing a variable, possibly storing the variable on the trail stack, and storing the value in the variable’s location. When it is know at compile-time that a variable will be given a value in this fashion, it is faster to create an uninitialized variable by just reserving a memory location and writing to this location at the point where unification would have taken place.

There has been much theoretical work on global analysis for Prolog, but few implementations. The Aquarius compiler shows that a simple dataflow analysis scheme is beneficial. Van Roy concludes that his compiler’s techniques suffice to remove much of the overhead of the features of logic programming when they are not used.

### 2.5. The Berkeley Abstract Machine (BAM) Processor

A criticism of many Prolog architectural projects is their focus on designs specifically developed for Prolog and the inability of such special-purpose processor design to keep pace with technical improvements in general-purpose processor development. The philosophy in the design of the Berkeley Abstract Machine (BAM) [7] is that of extending a general-purpose architecture to support Prolog without compromising general-purpose performance by providing support through compiler optimization and essential low-level operations. The development of the BAM processor and the Aquarius compiler [55] proceeded simultaneously with this shared philosophy so that the features of one complement the features of the other. In the BAM, most Prolog specific operations are done satisfactorily in software. Only a crucial set of features need be supported by the architecture to achieve efficient Prolog performance. Figure 2.13 shows the organization of the BAM processor that supports segmented virtual addresses, separate off-chip instruction and data caches, synchronized opcode and execution pipelines, and an internal opcode expansion scheme. Important features added for Prolog are instructions to support the BAM execution model, logic for tag support, and a double-word data port to memory.

BAM instructions are thirty-two bits with a six-bit opcode and fixed source register format that usually execute in a single cycle. Data objects are standard thirty-two-bit words. Although a word’s four most significant bits

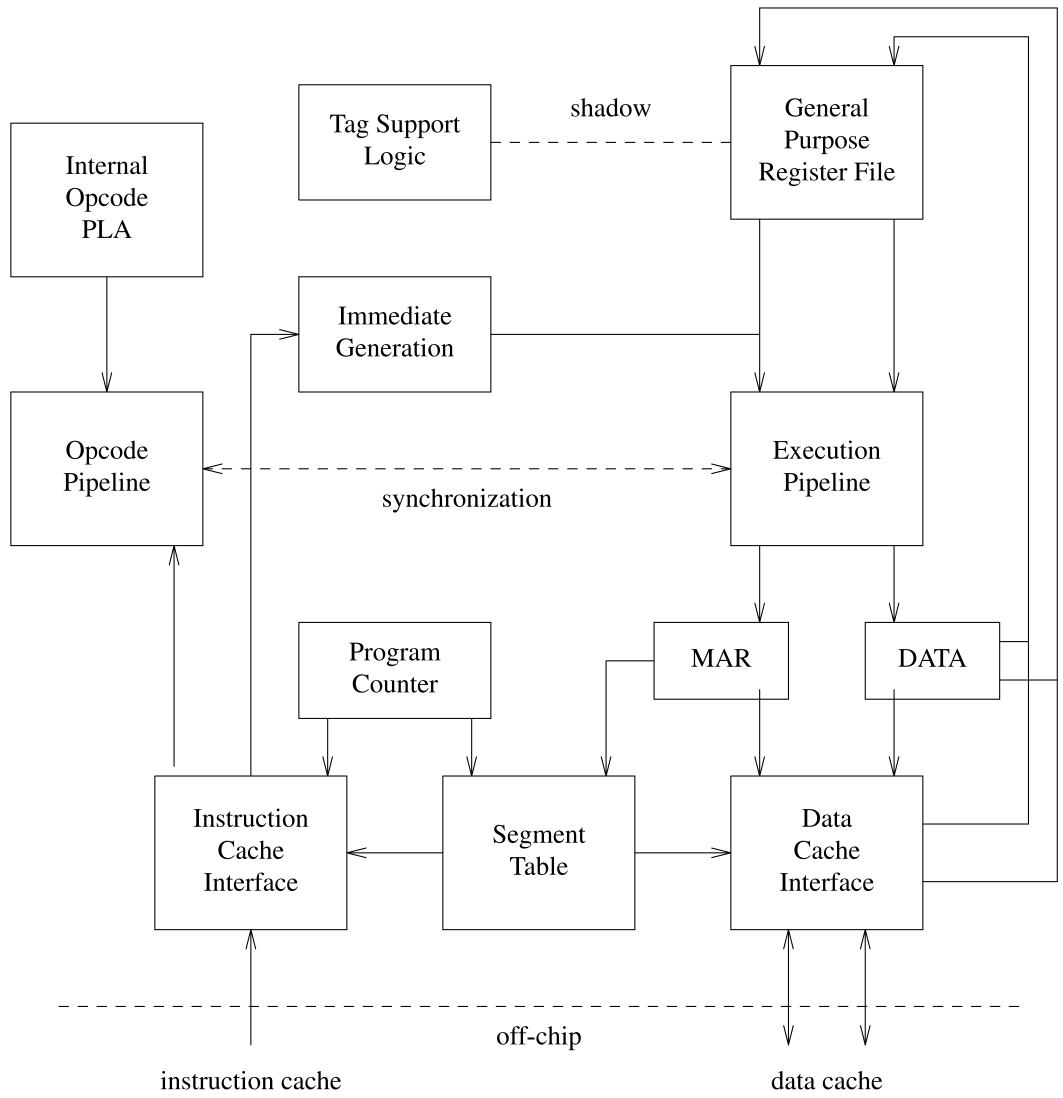


**Figure 2.13. BAM Processor Organization**

are used as a tag for Prolog objects, the distinction between tag and value fields is generally not recognized by the architecture. Arithmetic computation and address calculations use the entire data word in order to conform to general-purpose practices. BAM's general-purpose instructions include arithmetic and logic operations, conditional branches through separate compare and delayed branch instructions, and delayed load and store operations. Instructions following delayed branch instructions may be annulled if the branch is taken, or annulled if the branch is not taken, or always executed.

The BAM includes Prolog inspired instructions that are not often present in general-purpose processors but which can still be used for general computation. Small immediate constants can be loaded, stored, or used in a comparison. A bit-pattern of four zeroes for positive integers and four ones for negative integers allows that sign extending a small immediate value returns a properly tagged Prolog constant. Load immediate is used for creating integers and atoms. Store immediate is an optimization of a load immediate followed by a store and is used to bind an atom to a variable. Single-cycle, double-word load, store, push, and pop operations support Prolog's large memory bandwidth requirements and stack memory accessing pattern allowing compound term creation in a minimum number of cycles. An unsigned maximum instruction assists the management of the environment and choice point address pointers.

BAM's remaining instructions are tailored to specific requirements of Prolog. These support creation of unbound variables and compound terms, type checking, dereferencing, unification of atoms, and arithmetic overflow. Variable and compound term creation is assisted by load effective address instructions that calculate an address and then replace the most significant four bits with an appropriate tag. Type checking through single-cycle compare and branch-on-tag instructions allow the replacement of shallow backtracking with a conditional branch on an argument's tag. A three-way branch based on a single register's tag selects between unbound variables, specific immediate values, and all other tags to support unification. A second three-way branch instruction supports unification by selecting on the tags of two registers in directions for a variable/nonvariable pair, a nonvariable/variable pair, and all other pair combinations. Dereference is implemented as a single instruction that is expanded into a sequence of internal opcodes as explained below. This reduces static code size, allows dereference memory reads to be pipelined, and results in a tighter loop than the equivalent assembly code. A single-cycle unify-immediate instruction binds an atom to a variable if the variable is unbound and otherwise tests for equality. Arithmetic and compare instructions include versions that operate on the full thirty-two-bit words but trap when either of the sources or the result do not have integer tags.

Instruction execution is controlled by an opcode pipeline that operates in parallel with the execution pipeline. Each stage of the opcode pipeline decodes the opcode associated with that stage of the execution pipeline. Multicycle instructions, conditional instructions, and operating system support are implemented using internal opcodes. The internal opcodes of multicycle instructions are fetched from a PLA and inserted into the opcode pipeline. When an internal opcode is inserted, no instruction is fetched during that cycle. A single external opcode can invoke a sequence of internal opcodes to provide for often used complex operations.

Two sets of thirty-two registers include a general-purpose register set for procedure argument passing, temporary storage, and stack pointers, and a special register set that provides the processor status word, program counter, segment mapping table, and cache interface registers. The general-purpose register file has two read ports (a single-word port and a double-word port) and two write ports (both single-words).

The BAM uses segmented virtual addresses. A segment table maps the most significant six bits of an address to a twelve-bit value that is prepended to the remaining twenty-six address bits to form a thirty-eight-bit virtual address. The primary motivation for segmented virtual addressing is to map the thirty-two-bit address of a Prolog object to the proper BAM memory area. The scheme allows different Prolog data types to be mapped to the same virtual address segment and a given data type to be placed in one of several memory areas. Variable, list, and structure pointers use one virtual segment. Environments, choice points, the trail stack, and a symbol table are mapped to their own segments.

The processor design is tightly coupled with the cache design. The use of off-chip caches allows on-chip area to be used for enhancing architectural operations and allows fast, dense static RAM chips for large caches. Protection violation, consistency checks, and address tag comparison are computed on-chip to speed cache accesses.

The BAM project claims to have met their goal of achieving high-performance logic programming with a minimal set of extensions without compromising the performance of a general-purpose architecture. They identify tagged-immediate support, segment mapping, double-word memory access, special logic for fast branch-on-tag, and multi-cycle instruction support as important Prolog specific features. Significant performance benefit is obtained from a majority of the special-purpose instructions, especially the push, dereference, tagged-pointer creation, and tag branching instructions. The BAM project demonstrates that one can extend a general-purpose architecture to include explicit support for symbolic languages with a modest eleven percent increase in chip area and yet attain a significant seventy percent performance benefit.

# CHAPTER 3

# Methodology

We present our research methodology, the tools we have developed to assist our work, and a new suite of Prolog benchmarks. We discuss the static code characteristics, dynamic code characteristics, and memory usage of our benchmarks and compare these characteristics with those of other benchmarks cited in the literature.

## 3.1. Methodology and Tools

Our research is driven by the compiler and simulator for the load-store architecture we have developed to form the Simple Instruction Set Machine for Prolog Execution (SIMPLE) system. The SIMPLE compiler translates Prolog into SIMPLE instructions as diagrammed in figure 3.1. The front-end of the compiler was obtained from Berkeley. It is Van Roy's original PLM compiler [62] that translates Prolog into a WAM instruction set. We use the first three modules of Van Roy's compiler. An input module reads a Prolog program and filters it into a group of clauses for each predicate. A clause compilation module compiles each of the individual clauses. A procedure compilation module links together the blocks of WAM code for each of the clauses of a predicate into a WAM procedure. The PLM compiler uses an internal format that corresponds to WAM operations. The back-end of our compiler takes the internal WAM representation, expands it to sequences of SIMPLE instructions, and links in a library of internal routines.

The compiler's back-end works on one procedure at a time passing the procedure's representation through several stages. First, each of the WAM operations are expanded to a sequence of SIMPLE instructions that are represented in a new internal format. This expanded code contains references to Prolog's fundamental operations. The next stage expands these references to in-line SIMPLE instructions (for bind, trail, dereference, and decdr operations) or generates procedure calls to fundamental-operation routines (for general unification, fail, detrail, and evaluate operations). Eighty-two sequences of SIMPLE instructions were developed to emulate the variations of WAM operations. After expanding the fundamental operations, these sequences average fourteen instructions in length

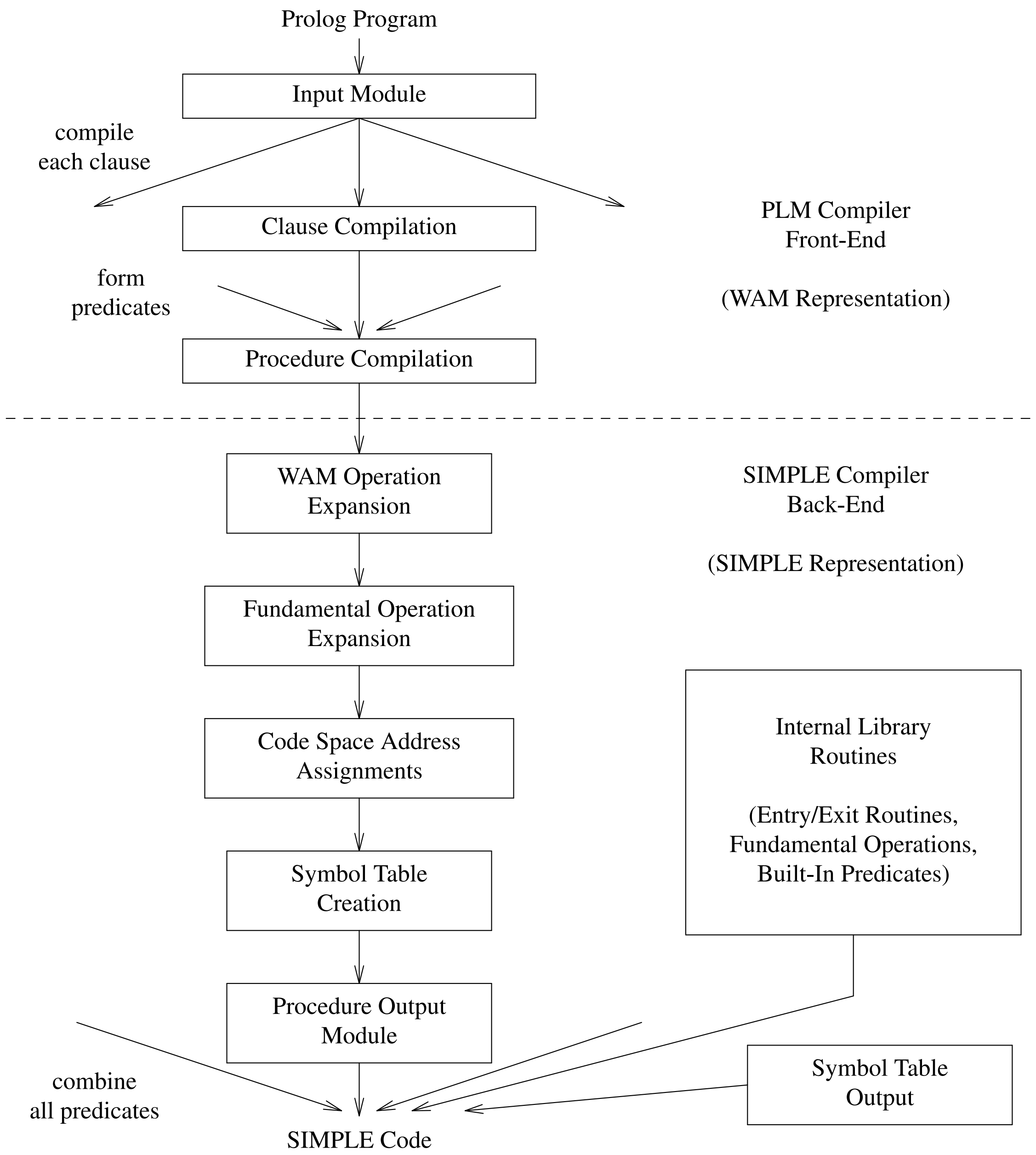


**Figure 3.1. SIMPLE Compiler Organization**

ranging from a single instruction to eighty-one instructions as shown for the major variations of WAM operations in table 3.1. The longest and most complex sequences are the unify operations that must handle special-case unification for both read and write modes of both list and structure elements. The get operations contain the next longest

| WAM Operation(s) | SIMPLE Instructions |
|---|---|
| get_variable | 1 |
| get_value | 25 |
| get_integer, get_atom, get_nil | 32 |
| get_structure | 43 |
| get_list | 39 |
| put_variable | 5 |
| put_value | 1 |
| put_integer, put_atom, put_nil | 2 |
| put_structure | 7 |
| put_list | 2 |
| unify_variable | 51 |
| unify_value | 71 |
| unify_integer, unify_atom | 81 |
| unify_nil | 43 |
| unify_cdr | 14 |
| call | 4 |
| execute | 2 |
| escape | 4 |
| proceed | 2 |
| allocate | 13 |
| deallocate | 3 |
| try_me_else, try | 24 |
| retry_me_else, retry | 3 |
| trust_me_else, trust | 4 |
| cut | 15 |
| switch_on_term | 12 |
| switch_on_constant | 11 |
| switch_on_structure | 15 |

**Table 3.1. SIMPLE Static Instruction Sequence Length of WAM Operations**

sequences. In-line expansion of the fundamental dereference, trail, and decdr operations contribute significantly to the length of the sequences of these two groups and to the switch operation sequences. Choice point creation and environment creation operations require a fair number of SIMPLE instructions to build and initialize their structures. The majority of WAM operations are implemented by a few to several SIMPLE instructions.

The next compiler stages assign addresses to each procedure's code, fill in branch target-destination fields, and record symbol table information for atoms and functors. The internal representation of procedure code is output as SIMPLE instructions in a human-readable assembly language format.

After all procedures have been processed, a library of internal routines is linked into the SIMPLE code. The library consists of thirteen hundred SIMPLE instructions distributed as shown in table 3.2. The library contains several procedures for program initialization, successful program completion, and error reporting. It contains routines to execute the fundamental operations of general unification, fail and detrail (combined in a single routine), and evaluate. Twenty-six internal routines perform arithmetic assignment and comparison, checking for term equivalence, classification of terms, and input/output to assist the forty built-in predicates supported by the system. (Several built-in predicates are implemented as one or two in-line instructions or are transformed to WAM operations during front-end processing.) The complexity of the longest of the library routines (general unification, evaluate, term equivalence, write) arises from their need to react to the different cases for arbitrary type terms that may be passed to them, their need to extract and dereference smaller components from nested structures, and their need to recursively manage each extracted component. Comparing the length of the general unification operation to the compiled unification operations indicates the degree of optimization that is achieved by the WAM. Many of the

| Internal Library Routine(s) | SIMPLE Instructions | |
|---|---|---|
| start-up and exit code | 39 | |
| error reporting | 94 | |
| initialization/completion code | | 133 |
| general unification | 166 | |
| fail and detrail | 23 | |
| evaluate | 102 | |
| total fundamental operations | | 291 |
| is | 45 | |
| arithmetic comparison (6) | 23 | |
| term equivalence (2) | 111 | |
| term classification (11) | 13 | |
| get (2) | 32 | |
| put (2) | 13 | |
| write | 161 | |
| total built-in predicates | | 878 |
| total | | 1,302 |

**Table 3.2. SIMPLE Static Code Size of Internal Library Routines**

library routines include instructions to communicate through a procedure call mechanism for their initial and recursive calls through use of the push-down list. Many of the built-in predicate routines contain in-line expanded dereference operations.

The last stage of the compiler outputs symbol table information to be placed at the top of data space.

Our base implementation is founded on building a WAM model with general-purpose architectural features. The compiler must implement specifics of the WAM. The compiler represents a Prolog object as a single word treating its first eight bits as the object's tag that includes type, subtype, and cdr-coding information. General-purpose registers are assigned the functions of ten WAM address pointers and eight argument registers. Several other registers are used to store temporary values or as accumulators. The four WAM stacks are placed in a non-partitioned, contiguous main memory below an area containing the symbol table.

The SIMPLE instruction set includes several operations and three addressing modes structured as a general-purpose, load-store instruction set architecture as shown in table 3.3. Operations include register-to-register moves, loads of an immediate value or a memory word's contents to a register, stores of an immediate value or a register's contents to memory, two-operand integer addition and subtraction, logical “and” and “or”, arithmetic shift left, and arithmetic shift right operations, unconditional and conditional branches and jumps, and input, output, and system call instructions. Operand types include immediate values, register operands, and memory operands. Memory operands are addressed by specifying a register whose contents are added to a specified immediate value to give the location of the operand in memory. Memory addressing is only available in loads and stores. Branch instructions compute their target address as an offset from the current instruction. Jumps specify their target as an absolute address. All instruction and data areas are thirty-two-bit word addressable.

The SIMPLE simulator reads and executes a SIMPLE assembly language program and generates statistics and traces of the execution. To further facilitate our work, the simulator accepts a richer language than we have described and the SIMPLE compiler generates a richer output. The compiler inserts references to WAM operations, run-time flags, and statistic generating instructions in the code it produces. This additional functionality increases the measurements we are able to collect and allows the study of many performance issues, including optimizations of the compiled code, without continuous need to rebuild the compiler and recompile the benchmarks. Many of the

| Opcode | Operands | Description |
|---|---|---|
| MOV | Address, Reg | load register from memory |
| MOV | Reg, Address | store register to memory |
| MOV | Imm or Reg, Reg | move immediate or register to register |
| ADD | Imm or Reg, Reg | add immediate or register to register |
| SUB | Imm or Reg, Reg | sub immediate or register to register |
| AND | Imm or Reg, Reg | bitwise-and immediate or register to register |
| ORR | Imm or Reg, Reg | bitwise-or immediate or register to register |
| SHL | Imm or Reg, Reg | logical shift-left immediate or register bits |
| SHR | Imm or Reg, Reg | logical shift-right immediate or register bits |
| BAL | Offset | branch always |
| BEQ | Reg, Reg, Offset | branch if registers equal |
| BNE | Reg, Reg, Offset | branch if registers not equal |
| BGT | Reg, Reg, Offset | branch if registers greater than |
| BLT | Reg, Reg, Offset | branch if registers less than |
| BGE | Reg, Reg, Offset | branch if registers greater than or equal |
| BLE | Reg, Reg, Offset | branch if registers less than or equal |
| JAL | Dest | jump always |
| JEQ | Reg, Reg, Dest | jump if registers equal |
| JNE | Reg, Reg, Dest | jump if registers not equal |
| JGT | Reg, Reg, Dest | jump if registers greater than |
| JLT | Reg, Reg, Dest | jump if registers less than |
| JGE | Reg, Reg, Dest | jump if registers greater than or equal |
| JLE | Reg, Reg, Dest | jump if registers less than or equal |
| INC | Reg | input character data |
| INI | Reg | input integer data |
| OTC | Imm or Reg | output immediate or register character data |
| OTI | Imm or Reg | output immediate or register integer data |
| SYS | Imm | system call |

**Table 3.3. SIMPLE Instruction Set**

results we present were generated directly by the simulator. Other results were obtained by tabulating information from raw data and traces.

### 3.2. Benchmarks

Eleven benchmarks, listed in table 3.4, were used in our work. The benchmarks cover the areas of artificial intelligence, natural language processing, compilers and symbolic translators, text processing, mathematical applications, and database applications, representing the more popular uses of Prolog. A variety of algorithms including

| Benchmark | Applications, Algorithms and Data Structures |
|---|---|
| cannibals | artificial intelligence search, list processing |
| characters | text processing, binary trees, updatable variables, searching and sorting |
| compiler | compilers, parsing, trees, difference lists |
| cryptoarithmetic | artificial intelligence search, arithmetic calculations |
| grammar | natural language analysis, parsing |
| hexconversion | arithmetic calculations |
| primes | number theory, list processing |
| queens | artificial intelligence search, list processing |
| routes | artificial intelligence planning, database operations |
| rowreduce | linear algebra, matrix operations |
| statistics | database operations, searching and sorting |

**Table 3.4. Benchmark Summary**

binary search, quick sort, table lookup, and matrix operations, and a variety of data structures including linear lists, difference lists, binary trees, and matrices are contained in the code.

*Cannibals* is a variation of the missionaries and cannibals problem. It is an extension of the solution developed in Covington, Nute, and Vellino [98] using an incremental generate and test search strategy. The *characters* benchmark determines the frequency of use of characters within a text file. It includes the creation and search of two binary trees [99] and a technique to implement updatable variables adapted from a suggestion by O'Keefe for implementing "updatable arrays" [100]. The *compiler* benchmark is expanded from the Algol-like compiler described in Warren's thesis [38] and run with compilations of twelve short procedures. *Cryptoarithmetic* solves cryptoarithmetic problems utilizing an incremental generate and test search strategy [101]. *Grammar* is a natural language parser based on a small context-free grammar [20]. *Hexconversion* reads a file of decimal numerals and produces a conversion table showing each decimal numeral and its equivalent hexadecimal representation. The *primes* benchmark uses the Sieve of Eratosthenes algorithm [102] to generate a table of prime numbers. It is adapted from the solution given by Clocksin and Mellish [13]. *Queens* generates all solutions to the eight-queens problem [99]. Its search strategy improves upon the generate and test technique through use of an agenda. The *routes* benchmark plans routes between different cities based on different criteria using a database of highway information, bus routes, and flight information. *Rowreduce* uses the technique of row reduction to solve a series of sets of linear equations. The *statistics* benchmark executes a sequence of queries to a small database performing several selections, projections, joins, arithmetic operations, and sorts.

Mostly, our benchmarks were derived from examples appearing in textbooks and recent literature. Care was taken to choose and develop code with a distinct declarative programming style, a mature approach to Prolog programming, and an efficient implementation of the underlying algorithm.

### 3.2.1. Static Benchmark Characteristics

Our benchmarks are of a small to medium size. They contain from one to two hundred lines of source code. Several static characteristics of the benchmarks are listed in table 3.5. The table shows the number of unique predicates and total clauses in each benchmark. The smallest benchmark contains seven predicates and fourteen clauses. The largest contains thirty-nine predicates and 133 clauses. The table shows the number of WAM operations and SIMPLE instructions in each program. The benchmarks range from five thousand SIMPLE instructions (two hundred WAM operations) to forty thousand SIMPLE instructions (fifteen hundred WAM operations). The average benchmark compiles to seven hundred and sixty WAM operations or nineteen thousand SIMPLE instructions. In

| | Prolog Code | | WAM | SIMPLE |
|---|---|---|---|---|
| Benchmark | Predicates | Clauses (Facts) | Operations | Instructions |
| cannibals | 16 | 28 | 807 | 22,988 |
| characters | 14 | 30 | 348 | 8,959 |
| compiler | 39 | 133 (63) | 1,471 | 38,778 |
| cryptoarithmetic | 11 | 24 | 458 | 12,761 |
| grammar | 29 | 100 (70) | 993 | 33,067 |
| hexconversion | 13 | 36 (10) | 329 | 6,606 |
| primes | 7 | 14 | 172 | 4,754 |
| queens | 7 | 14 | 201 | 5,471 |
| routes | 19 | 98 (55) | 1,161 | 26,392 |
| rowreduce | 28 | 60 | 980 | 20,584 |
| statistics | 33 | 136 (84) | 1,429 | 30,179 |
| arithmetic mean | | | 759 | 19,140 |
| total | 216 | 673 | 8,349 | 210,536 |

**Table 3.5. Static Benchmark Characteristics**

total, we simulate over two hundred thousand static SIMPLE instructions to emulate over eight thousand WAM operations.

SIMPLE programs include code for the programmer-defined predicates linked with the internal library routines. Table 3.6 shows the library contributes from three percent to twenty-seven percent to the static code size in our benchmarks. On average, the library accounts for eleven percent of SIMPLE code. The table also shows how SIMPLE instructions are distributed among the facts and rules of each benchmark.

Most of the benchmark code is algorithmic in nature. The clauses are predominantly rules. The exceptions are the grammar, statistics, routes, compiler and hexconversion benchmarks where a significant percentage of the clauses are facts. Fact-intensive programs present a difficulty in the WAM model in that a seemingly small Prolog program with many facts produces a surprisingly large SIMPLE program. Despite the simplistic appearance of facts, a significant portion of compiled code is generated from the head of a clause. Fact-intensive programs are also interesting in that they predominantly portray the characteristics of the most important compiled unification operations. A fact-intensive benchmark is more likely to resemble a "pure" Prolog program.

| Benchmark | Percentage of SIMPLE Instructions | | |
|---|---|---|---|
| | Programmer-Defined Predicates | | Internal Library |
| | Rules | Facts | Routines |
| cannibals | 94.4 | 0.0 | 5.6 |
| characters | 84.7 | 0.0 | 14.3 |
| compiler | 71.3 | 25.4 | 3.3 |
| cryptoarithmetic | 90.0 | 0.0 | 10.0 |
| grammar | 19.2 | 76.9 | 3.9 |
| hexconversion | 62.7 | 17.9 | 19.4 |
| primes | 73.1 | 0.0 | 26.9 |
| queens | 76.6 | 0.0 | 23.4 |
| routes | 64.4 | 30.8 | 4.8 |
| rowreduce | 93.8 | 0.0 | 6.2 |
| statistics | 40.3 | 55.5 | 4.2 |
| arithmetic mean | 70.0 | 18.8 | 11.1 |

**Table 3.6. Rules, Facts, and Library Static Code Percentages**

In the WAM model, facts are treated as rules and are compiled into SIMPLE instructions. Fact procedures are predominantly sequences of get and unify operations. Each get and unify operation performs dereferencing, perhaps decdring, determines and reacts to the mode of a term passed to its procedure, and then creates an object to which the term is instantiated or verifies that the term matches some object. Due to the complexity of get and unify operations, programs that contain a large number of facts, (for example, large database applications), compile to large SIMPLE programs. In appendix A, we present a technique to reduce the static size of fact-intensive programs at a modest cost in execution time.

Table 3.7 shows the benchmarks to exhibit a static SIMPLE instructions to WAM operations ratio ranging from twenty to thirty-three and averaging twenty-six. The high ratio of SIMPLE instructions to WAM operations is due to the WAM's ability to implicitly encode multiple operations and its implicit use of temporary registers and constant operands, implicit addressing to stack areas, and knowledge of the use of tag bits. Each operation and resource must be explicitly specified in SIMPLE code. By considering the size of WAM instructions on the PLM (one to six bytes) and knowing the static frequency of WAM operations for our benchmarks and by treating SIMPLE

| Benchmark | SIMPLE Instructions to WAM Operations | |
|---|---|---|
| | Static | Dynamic |
| cannibals | 28.5 | 40.3 |
| characters | 25.7 | 21.9 |
| compiler | 26.4 | 26.8 |
| cryptoarithmetic | 27.9 | 28.4 |
| grammar | 33.3 | 33.1 |
| hexconversion | 20.1 | 31.2 |
| primes | 27.6 | 18.6 |
| queens | 27.2 | 26.2 |
| routes | 22.7 | 35.6 |
| rowreduce | 21.0 | 25.9 |
| statistics | 21.1 | 26.8 |
| arithmetic mean | 25.6 | 28.6 |

**Table 3.7. Static and Dynamic SIMPLE Instruction to WAM Operation Ratios**

instructions as a four-byte word, we estimate the size of a program compiled into SIMPLE instructions to be twenty-five times the size of a program compiled into PLM instructions.

The SIMPLE to PLM static code-size ratio of twenty-five is greater than similar ratios reported by other researchers. The SPUR PLM implementation reports a SPUR to PLM static code-size ratio of seventeen [43]. Mills estimates LOW RISC code to be seven times the size as code on the PLM [6]. BAM code is only three times larger than PLM code [7]. Several factors have contributed to the larger ratio in our study including differences in the static distribution of WAM operations in our benchmark suite compared to benchmark suites of other groups, the existence of compiler optimization techniques and special-purpose architectural features in other studies that are not included in our base implementation, and an intentional reduction in code size through placement of common operations in software libraries and/or hardware.

Not surprisingly, the SPUR study comes closest to our calculation as its macroexpansion scheme is closest to our compilation scheme and its instruction set has comparatively small differences from our base instruction set. The SPUR implementation's ability to achieve a lower code-size ratio is due to its use of register windows for choice points, its inclusion of instructions that support tagged objects, and its placement of a greater number of common operations within a library. The use of register windows eliminates transfer instructions needed in choice point creation and backtracking because address pointers which are assigned within a register window are automatically saved and restored by a single register window shift instruction. Tagged-data support reduces instructions needed for the extraction and combination of tag and value fields of data objects. The SPUR implementation's library includes fundamental-operation routines, built-in predicate routines, and a few routines that emulate entire WAM operations. We have chosen to expand all WAM operations and most fundamental operations to in-line SIMPLE instructions.

The LOW RISC achieves a still lower static code-size ratio through compiler optimizations such as those that eliminate unnecessary trailing and dereferencing. It's lower-level, but WAM-derived instruction set helps to lower static code size. Instructions implicitly update status flags for Prolog-specific conditions that must be explicitly computed in SIMPLE. The LOW RISC also includes the use of register windows for choice point implementation.

The surprisingly small code size of the BAM is achieved through direct compilation to its machine instruction set using extensive compiler optimizations and an abstract machine model that the BAM's developers claim allows a finer granularity of optimization than the WAM. The BAM includes a fair number of Prolog-specific instructions. It's scheme of internal opcodes allows complex, multicycle operations to be achieved with single instructions.

The disadvantage of compiling Prolog to a lower-level instruction set of increased static code size is significantly reduced and reversed when an intelligent compiler exploits the exposure of lower-level details through both local and global optimizations. (An analysis of code size reduction through global optimization techniques for our benchmark suite is presented in the next chapter.) Compiler optimization techniques, such as those in the compilers for the LOW RISC and BAM machines, are useful to other existing machines. Subsequent reductions in code size achieved through architectural enhancements are less applicable to existing machines. Hardware features that reduce code size in each of the above implementations are not common on all machines and some are very specific to Prolog.

### 3.2.2. Dynamic Benchmark Characteristics

Dynamic characteristics for the benchmarks are shown in table 3.8. The table shows the number of procedure

| Benchmark | Procedure Calls | | WAM Operations | SIMPLE Instructions |
|---|---|---|---|---|
| | Programmer-Defined | Built-In Predicate | | |
| cannibals | 22,628 | 37,893 | 264,582 | 10,660,841 |
| characters | 19,959 | 24,498 | 189,192 | 4,136,390 |
| compiler | 11,571 | 15,036 | 124,438 | 3,329,389 |
| cryptoarithmetic | 24,440 | 48,059 | 364,398 | 10,331,487 |
| grammar | 9,282 | 3,511 | 95,298 | 3,149,615 |
| hexconversion | 10,278 | 23,675 | 180,291 | 5,622,912 |
| primes | 5,432 | 9,320 | 111,170 | 2,064,403 |
| queens | 19,540 | 29,206 | 267,578 | 7,000,636 |
| routes | 19,266 | 4,873 | 154,631 | 5,501,016 |
| rowreduce | 11,266 | 14,668 | 164,292 | 4,247,533 |
| statistics | 14,572 | 3,526 | 94,553 | 2,534,097 |
| arithmetic mean | 15,294 | 19,479 | 182,766 | 5,325,301 |
| total | 168,234 | 214,265 | 2,010,423 | 58,578,319 |

**Table 3.8. Dynamic Benchmark Characteristics**

calls to programmer-defined predicates, the number of executed built-in predicate routines, the number of WAM operations emulated, and the number of SIMPLE instructions executed. Comparing the number of programmer-defined procedures executed to the number of built-in predicates executed shows the two values to be of the same order of magnitude for most benchmarks. Overall, twenty-seven percent more built-in predicates than programmer-defined predicates are executed suggesting that a significant amount of time is spent within these routines and indicating their importance in a Prolog implementation. The benchmarks execute between one-hundred thousand and three-hundred and sixty thousand WAM operations (averaging two hundred thousand WAM operations) and between two million and eleven million SIMPLE instructions (averaging five million SIMPLE instructions). In total, just under sixty million SIMPLE instructions are executed in our simulations.

In table 3.9, we show how the SIMPLE instructions executed are distributed among the programmer-defined rules, programmer-defined facts, and the library routines. About half of the instructions executed in the grammar benchmark are within fact code. One quarter of the instructions executed in the compilers and routes benchmarks are within fact code. These benchmarks make use of a large portion of the facts represented. The other fact-

| Benchmark | Percentage of SIMPLE Instructions | | |
|---|---|---|---|
| | Programmer-Defined Predicates | | Internal Library Routines |
| | Rules | Facts | |
| cannibals | 32.1 | 0.0 | 67.9 |
| characters | 73.2 | 0.0 | 26.8 |
| compiler | 35.3 | 24.7 | 40.0 |
| cryptoarithmetic | 49.7 | 0.0 | 50.3 |
| grammar | 22.9 | 49.3 | 27.8 |
| hexconversion | 31.6 | 5.6 | 62.8 |
| primes | 52.7 | 0.0 | 47.3 |
| queens | 45.5 | 0.0 | 54.5 |
| routes | 36.1 | 26.3 | 37.6 |
| rowreduce | 52.4 | 0.0 | 47.6 |
| statistics | 59.5 | 12.2 | 28.3 |
| arithmetic mean | 44.6 | 10.7 | 44.6 |

**Table 3.9. Rules, Facts, and Library Dynamic Instruction Percentages**

intensive benchmarks spend a less significant amount of time within fact code. The benchmarks execute from twenty-five to seventy percent of instructions from the internal library. For most benchmarks, almost fifty percent of the instructions executed are within the library. The exceptions are the characters, grammar, and routes benchmarks that spend much time creating, manipulating, and searching through data structures and the statistics benchmark that spends its time sorting data and matching fields to perform database selections. Programs that mostly create and manipulate symbolic data make less use of library routines because they spend more time in compiled unification code.

The impact of the large amount of instructions executed within internal routines and especially within built-in predicate routines has largely been unexplored. The most common benchmarks include few built-in predicates and concentrate on a small subset of "pure" Prolog. While in some languages and applications, library functions serve to add functionality that is apart from the underlying algorithm (for example, operating system support) or contain functionality that is either important to only some applications or easily optimized independent from the context of use (for example, advanced mathematical functions), internal routines and built-in predicates serve a more fundamental role in Prolog both in symbolic applications (comparison of terms, classification of terms, forced backtracking, explicit unification, negation), and elsewhere (simple arithmetic, numerical comparison). Often a seemingly simple built-in predicate is more costly than expected. Evaluation or comparison of arithmetic expressions often takes one to two hundred SIMPLE instructions because of Prolog's representation of arithmetic expressions and the need for internal routines to handle the generic case. The input of a simple term averages twenty-five SIMPLE instructions. In addition to reading a value, the input routine must create the tagged Prolog object and link it to a dereferenced variable. As with unification, many generic built-in predicate operations may be optimized when the context in which they are executing has been determined by compile-time analysis. It is ironic that several scientific benchmark suites gain significant performance improvements that amount to much smaller improvements in real applications by optimization of one or two library routines while in Prolog were the optimization of a set of built-in predicates can have significant impact in many applications, this aspect of analysis and performance improvement has largely been ignored. Many reported research results have a reduced impact in real applications because research has predominantly concentrated on only a portion of the application's code.

Back in table 3.7 we have included the dynamic ratios of SIMPLE instructions to WAM operations to reemphasize the efficiency of the encoding of WAM operations for Prolog. On average, twenty to forty SIMPLE instructions are needed to emulate each WAM operation.

### 3.2.3. Locality of Program Execution

The measurements of the size and activity of the internal library routines indicate that forty-five percent of instructions executed (as shown in table 3.9) are contained within eleven percent of the static code (as shown in table 3.6). Figure 3.2 shows the locality of execution of code space at the instruction level for the combined benchmarks. The figure shows that the most-active five percent of code is responsible for eighty-four percent of the instructions executed and the ten percent most-active code accounts for ninety-four percent of the instructions executed. This high degree of locality is encouraging in a language which in practice lacks a loop control structure and which after compilation lacks small, highly-repetitive loops. The majority of branches within a Prolog program

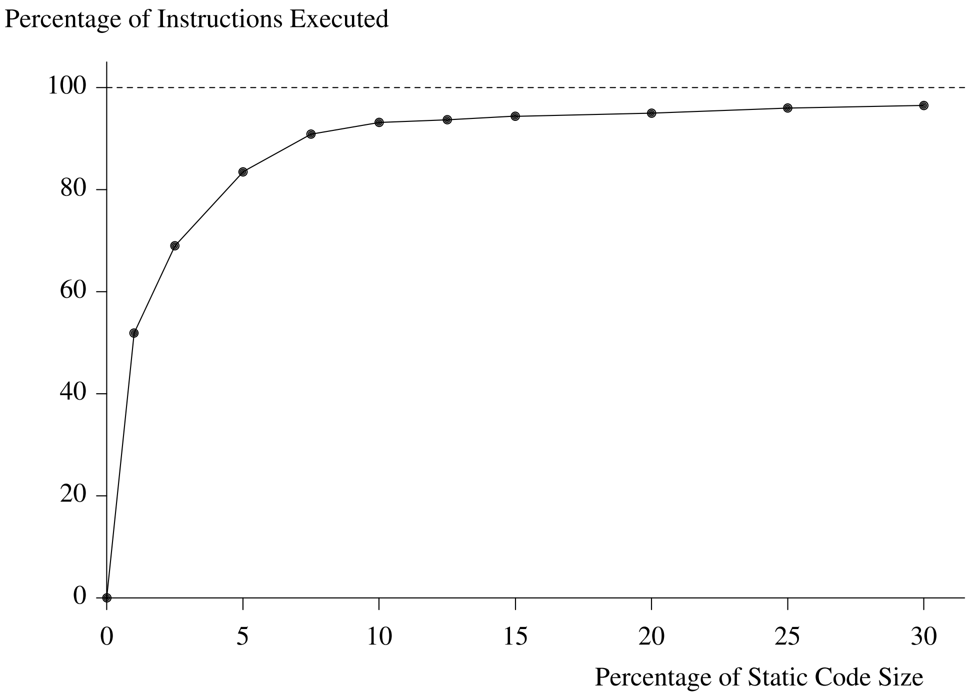


**Figure 3.2. Percentage of Instructions Executed within Most-Active Code**

contain forward target-addresses to skip beyond non-applicable cases in compiled unification code. Some backward branching exists within the code for the dereference and decdr fundamental operations but these loops are almost never repeated more than a couple of times. The major causes of code repetition are multiple calls to a procedure, recursion within a procedure, and backtracking in the programmer-defined code and the recursive library routines of general unification and evaluate and the built-in predicates of term equivalence and write. Code repetition takes place over a greater span of instructions. Procedures average a size of nine hundred SIMPLE instructions. These larger "loops" suggest that Prolog has a lower temporal instruction locality than languages with many highly-repetitive, small inner loops. Tick presents an extensive report on Prolog memory-referencing behavior predominantly concentrating on Prolog's performance with various cache and buffer configurations [103, 104]; however, we are aware of no report that compares more fundamental measures of memory-referencing behavior of Prolog to other languages.

Table 3.10 shows that a surprisingly large amount of the code in the benchmarks remains unexecuted after the

| Benchmark | Percentage of Static Code Size | | |
|---|---|---|---|
| | Programmer-Defined Unexecuted SIMPLE Instructions | Internal Library Unexecuted SIMPLE Instructions | Total Unexecuted SIMPLE Instructions |
| cannibals | 69.4 | 3.8 | 73.2 |
| characters | 57.6 | 10.0 | 67.6 |
| compiler | 52.3 | 2.2 | 54.5 |
| cryptoarithmetic | 62.1 | 7.2 | 69.3 |
| grammar | 64.8 | 3.1 | 67.9 |
| hexconversion | 67.6 | 13.8 | 81.4 |
| primes | 62.7 | 21.0 | 83.7 |
| queens | 49.4 | 16.9 | 66.3 |
| routes | 47.3 | 3.1 | 50.4 |
| rowreduce | 52.1 | 4.0 | 56.1 |
| statistics | 53.7 | 3.3 | 57.0 |
| arithmetic mean | 58.1 | 8.0 | 66.1 |

**Table 3.10. Unexecuted Code Percentages**

simulations complete. Disregarding the small portion of code due to unexecuted internal library routines that can be excluded at compile time, between forty-seven and seventy percent of an individual benchmark's code are unexecuted instructions. Unexecuted code results because input data to a program does not effect all situations that the program is designed to handle. Unexecuted code results because each benchmark procedure does not use the full flexibility that the WAM operations allow. The large percentage of unexecuted code is indicative of the high degree to which global optimization is of use to the WAM model in Prolog compilation. As we present optimizations within chapter 4, we show where unexecuted code exists within the benchmark code and how it can be reduced through global optimization.

Unexecuted code has at least three effects that decrease performance. First, the unused code reduces instruction spatial locality, especially if it is spread throughout the active code. This reduces the effectiveness of instruction caching. Second, unexecuted code causes an additional increase in the static code size because additional instructions are included to branch around dead code that would not be included if it were determined that the code was not necessary at compile time. Third, these extra instructions increase execution time as they compute and make decisions at run-time that the compiler might have been able to determine using more intelligent optimization algorithms.

#### 3.2.4. Memory Reference Characteristics

The number of memory references, register references, and immediate value references for each benchmark are shown in table 3.11. The number of memory references ranges from one million to two-and-a-half million references, averaging twelve-hundred thousand references per benchmark. The number of register references average sixty-five hundred thousand while the number of immediate references average four million per benchmark. The combined benchmarks make 134,000,000 references to memory, 72,000,000 references to registers, and 42,000,000 references to immediate values during the simulations.

Table 3.12, shows the usage of the different memory areas. Memory size requirements for the majority of the benchmarks are under two thousand words although a few of the benchmarks require significantly more memory. The area demanding the largest space requirements is the global stack. The global stack for most benchmarks requires under a thousand words with the exception of benchmarks that create large data structures and store many

| Benchmark | Memory References | Register References | Immediate References |
|---|---|---|---|
| cannibals | 2,206,512 | 12,850,068 | 8,266,197 |
| characters | 920,991 | 4,932,716 | 3,043,516 |
| compiler | 825,201 | 4,022,031 | 2,366,235 |
| crytoarithmetic | 2,502,242 | 12,733,166 | 7,150,809 |
| grammar | 705,116 | 3,909,853 | 2,297,204 |
| hexconversion | 1,312,213 | 7,002,208 | 3,920,947 |
| primes | 473,612 | 2,525,865 | 1,441,312 |
| queens | 1,380,534 | 8,771,636 | 5,155,456 |
| routes | 1,542,561 | 6,561,308 | 3,696,911 |
| rowreduce | 885,359 | 5,260,607 | 3,100,093 |
| statistics | 637,845 | 3,005,819 | 1,797,095 |
| total | 13,392,186 | 71,575,277 | 42,235,775 |

**Table 3.11. Benchmark Operand Reference Characteristics**

| Benchmark | Words of Memory | | | | | |
|---|---|---|---|---|---|---|
| | Atom Space Usage | Global Stack Usage | Local Stack Usage | Trail Stack Usage | PDL Stack Usage | Total Memory Usage |
| cannibals | 227 | 5,562 | 1,764 | 122 | 11 | 7,686 |
| characters | 85 | 14,070 | 427 | 2,688 | 6 | 17,276 |
| compiler | 523 | 1,641 | 3,023 | 524 | 10 | 5,721 |
| cryptoarithmetic | 42 | 245 | 777 | 67 | 11 | 1,142 |
| grammar | 515 | 434 | 738 | 202 | 21 | 1,910 |
| hexconversion | 66 | 696 | 1,156 | 31 | 17 | 1,966 |
| primes | 65 | 20,552 | 61 | 997 | 6 | 21,681 |
| queens | 42 | 719 | 392 | 23 | 6 | 1,182 |
| routes | 302 | 216 | 669 | 112 | 39 | 1,338 |
| rowreduce | 49 | 12,504 | 153 | 544 | 9 | 13,259 |
| statistics | 726 | 980 | 89 | 253 | 11 | 2,059 |

**Table 3.12. Benchmark Memory Usage Characteristics**

arithmetic expressions. The primes, characters, rowreduce, and cannibals benchmarks heavy global stack needs are due mostly to storage of arithmetic expressions. Most benchmarks require only several hundred words for their local stack area. The exceptions are compiler, cannibals, and hexconversion that contain a deep layering of choice points. A few hundred words is sufficient for the majority of the benchmarks. The push-down list requires only several words and at most a few dozen words depending on the size of the largest data structures that must be unified or compared and the size of the largest arithmetic expressions to be evaluated. The routes benchmark high PDL demand is caused by evaluation of large arithmetic expressions. The grammar benchmark uses general unification on a large sized data structure. Each benchmark needs atom space to hold its symbol table that on average contains two hundred and forty words.

We limit our discussion of memory characteristics in this report. The reader is referred to Tick's study for a more comprehensive report [103, 104].

### 3.3. Quality of the Benchmarks

An important and much needed contribution that we add to Prolog implementation research is a new benchmark suite. The benchmarks most often cited in the literature are the eight Warren benchmarks [38] and the six PLM benchmarks [5]. These benchmarks form the basis for results in the Tick, PLM, PSI, SPUR, and LOW RISC studies. Unfortunately, these benchmarks are small-sized Prolog programs. As a group, they do not contain much variety and do not contain a full complement of the use of the language. We felt these benchmarks were inadequate by themselves and chose to develop larger benchmarks that are more representative of the use of Prolog. Our programs more evenly spread the use of WAM operations. Our benchmarks contain a greater variety and use of built-in predicates. They favor the use of other built-in predicates over cuts to select between alternatives as is agreed to be the more correct programming practice. The search spaces in our programs are broader showing larger frequencies of use of choice point operations. Recently, a new set of benchmarks have appeared in the literature [104, 7] which we refer to as the BAM benchmarks. Within the BAM benchmarks are a few programs more similar in size statically or dynamically to our programs. Table 3.13 compares the static and dynamic instruction counts, memory reference count, and memory usage of the different benchmark suites.

The Warren benchmarks' use of built-in predicates is restricted to a fair number of cut operations. Only three of the PLM benchmarks make use of a variety of built-in routines. The static size of the Warren and PLM benchmarks average two hundred and just over one hundred WAM operations. Our benchmarks average seven hundred and sixty WAM operations in size. When the benchmarks from both of these groups are combined, they comprise

| Benchmark Suite | SIMPLE Instructions Static | SIMPLE Instructions Dynamic | Memory References | Memory Usage |
|---|---|---|---|---|
| Warren arithmetic mean | 7,860 | 35,272 | 5,454 | 500 |
| PLM arithmetic mean | 3,635 | 135,268 | 30,975 | 749 |
| BAM | | | | |
| chat_parser | 175,647 | 20,000,000 * | 3,700,000 * | - |
| meta_qsort | 15,358 | 649,068 | 173,877 | 17,625 |
| poly_10 | 18,457 | 6,515,099 | 1,260,811 | 123,708 |
| prover | 24,580 | 124,871 | 93,687 | 609 |
| simple_analyzer | 81,606 | 6,500,000 * | 1,200,000 * | - |
| tak | 2,448 | 19,802,950 | 4,166,700 | 1,168,295 |
| SIMPLE arithmetic mean | 19,140 | 5,325,301 | 1,217,472 | 6,838 |
| * estimated values | | | | |

**Table 3.13. Comparative Characteristics of Benchmark Suites**

less than forty percent of the static size of our benchmarks measured by SIMPLE instructions. The Warren and PLM benchmarks execute three thousand and ten thousand WAM operations and thirty-five thousand and 135,000 SIMPLE instructions on average. Our benchmark suite executes over twenty-five times the number of WAM operations with over fifty times the number of SIMPLE instructions. There is at least one and usually two orders of magnitude difference between the average dynamic instruction count of our benchmarks and the Warren and PLM programs. The Warren and PLM benchmarks fall short in their use of memory. Only two of the PLM benchmarks need more than one thousand words of memory. Only four of the Warren benchmarks need more than five hundred words of memory. This is in comparison to our typical benchmark's demand for two thousand words of memory, after excluding our memory-consuming benchmarks. The Warren benchmarks make an average of four thousand references to memory while the PLM benchmarks average twenty-one thousand memory references. Our benchmarks make almost eight times as many memory references. Overall, our benchmark suite contains larger programs, that execute more operations, require more memory, and make a greater number of memory references than both the Warren and PLM benchmark suites. They are representative of a greater diversity of applications and utilize a fuller set of the Prolog language.

The BAM benchmarks introduce six new programs. Three programs are less interesting. Two have an order of magnitude less instructions executed than our average benchmark (`prover`, `meta_sort`). One is on the same order of magnitude both statically and dynamically and has a high stack space demand (`poly_10`). Three programs are more interesting (`chat_parser`, `simple`, `tak`). With the possible exception of the chat_parser benchmark, our benchmarks are comparable in size, in execution time, and in the use of Prolog with the BAM benchmarks. Comparing the benchmark suites as wholes, our benchmarks exhibit a much higher consistency of quality. Two of the BAM benchmarks are of a larger static size. Chat is almost as large as our entire benchmark suite. The simple_analyzer is twice the size of our largest benchmark; however the number of instructions that are executed in simple_analyzer is less than than number of instructions executed by our average benchmark. Two of the BAM benchmarks execute more instructions. Chat executes less built-in predicates, but three times the number of programmer-declared predicates. Tak executes almost one-third the number of instructions as our entire suite; however tak is a very small program. Tak's large dynamic size and large use of the local stack is due to its inputs that force the program to recurse many times. The benchmark is less interesting because its recursion takes place over the same sections of code and in the same ways. Only the chat_parser benchmark is of a larger size and executes more instructions than our benchmarks. The characteristics of chat_parser are similar to our measurements of the grammar benchmark. Both these benchmarks are parsers of English sentences. They manipulate similar types of structures and both use built-in predicates infrequently.

# CHAPTER 4

# WAM-Level Benchmark Characteristics

We first examine the costs of executing WAM operations and Prolog's built-in predicate routines and operations fundamental to a WAM implementation. We then discuss and evaluate potential performance gain for enhancements at the abstract machine level obtained through mode analysis and global optimization.

### 4.1. Costs of WAM Operations and Built-In Predicate Operations

Dobry measured the dynamic frequencies of WAM operations and approximated the number of cycles that operations required on the PLM to obtain a rough estimate or "weight" of execution cost for WAM operations for the Warren and PLM benchmarks [5]. Although our benchmarks contain a greater variety and use of built-in predicates (that results in twice the frequency of escape operations), they share similar frequencies (though somewhat more evenly distributed) of static and dynamic distributions within the WAM operations with these benchmark suites. We have measured the exact cost of static WAM operation use and the exact costs of dynamic WAM operation and internal library routine use computed as percentages of SIMPLE instructions. We observe three moderate differences between our dynamic cost measurements and those of Dobry that are representative of the simplicity of the Warren and PLM benchmarks and the ability of the PLM to overlap operations within its microcoded instructions. Dobry finds the `put_value` operation, a simple argument register transfer, to be costly while this operation has a small cost in our measurements. The reverse is true with the more complex `get_constant` operation. The former operation and other simpler operations has a higher occurrence of use in the simpler benchmarks while the later and other more complex operations takes advantage of parallelism in the PLM datapath to decrease their cycles of execution. The benchmarks in Dobry's study have higher costs for the `switch_on_term` operation. The selection of clauses is more elaborate in our benchmarks. In the Warren and PLM benchmarks, the selection between clauses is often determined solely by whether the first procedure argument is an empty or non-empty list. In most cases, our cost measurements are in agreement with Dobry's weights.

Table 4.1 shows the static code size percentages for the classes of WAM operations. The table shows that eighty percent of static code consists of instructions for compiled unification. The smallest contribution of compiled unification instructions is still sixty-two percent. On average, unify operations comprise almost half of the static code reflecting the heavy use of lists and structures in most benchmarks and the length of the SIMPLE sequences needed to emulate them. The get operations comprise almost a quarter of the static code. The procedural, choice point, and switch operations contribute four, three, and one percent of static code.

Table 4.2 shows that compiled unification instructions account for forty-two percent of the instructions executed, predominantly split between the get and unify operations. The lowest dynamic use of compiled unification is twenty-four percent while the greatest dynamic use is fifty-eight percent. The benchmarks that most often use lists and structures (for example, characters and primes) execute more instructions within unify operations. The benchmarks that predominantly pass atoms as arguments (for example, hexconversion) execute more instructions within get operations. The WAM operation classes of choice point, procedural, and switch operations account for approximately seven, four and two percent of instructions executed.

| Benchmark | Percentage of SIMPLE Instructions | | | | | |
|---|---|---|---|---|---|---|
| | WAM Operations | | | | | |
| | Get | Put | Unify | Procedural | Choice Point | Switch |
| cannibals | 11.9 | 6.6 | 70.3 | 3.2 | 1.4 | 0.4 |
| characters | 16.5 | 12.1 | 46.3 | 4.2 | 4.3 | 0.7 |
| compiler | 23.0 | 8.3 | 58.5 | 2.7 | 2.7 | 1.1 |
| cryptoarithmetic | 14.7 | 12.0 | 55.8 | 3.1 | 2.7 | 0.4 |
| grammar | 27.3 | 4.5 | 58.5 | 1.5 | 2.5 | 1.2 |
| hexconversion | 25.5 | 6.8 | 36.6 | 4.4 | 3.9 | 1.2 |
| primes | 19.1 | 4.2 | 38.8 | 3.4 | 4.3 | 0.3 |
| queens | 12.0 | 12.2 | 41.9 | 4.3 | 2.9 | 0.7 |
| routes | 39.5 | 9.7 | 38.2 | 2.8 | 3.2 | 1.2 |
| rowreduce | 15.0 | 14.8 | 51.6 | 5.7 | 5.0 | 1.0 |
| statistics | 45.6 | 6.6 | 35.6 | 3.6 | 2.5 | 1.2 |
| arithmetic mean | 22.7 | 8.9 | 48.4 | 3.5 | 3.2 | 0.9 |

**Table 4.1. WAM Operation Static Code Size Percentages**

| Benchmark | Percentage of SIMPLE Instructions | | | | | |
|---|---|---|---|---|---|---|
| | WAM Operations | | | | | |
| | Get | Put | Unify | Procedural | Choice Point | Switch |
| cannibals | 5.4 | 4.2 | 14.2 | 2.5 | 5.0 | 0.6 |
| characters | 17.3 | 3.5 | 33.5 | 6.0 | 10.3 | 2.3 |
| compiler | 19.5 | 5.1 | 16.6 | 5.3 | 8.2 | 3.8 |
| cryptoarithmetic | 16.0 | 4.2 | 18.6 | 4.6 | 6.2 | 0.1 |
| grammar | 33.1 | 2.5 | 22.0 | 2.2 | 6.9 | 3.4 |
| hexconversion | 17.1 | 3.6 | 8.8 | 3.3 | 3.7 | 0.8 |
| primes | 12.8 | 3.5 | 22.2 | 4.8 | 9.0 | 0.2 |
| queens | 11.3 | 6.6 | 18.3 | 5.4 | 2.4 | 1.4 |
| routes | 29.9 | 2.4 | 13.4 | 2.1 | 10.9 | 3.7 |
| rowreduce | 13.9 | 6.8 | 19.3 | 4.9 | 5.7 | 1.8 |
| statistics | 21.9 | 2.9 | 29.6 | 3.1 | 12.8 | 1.4 |
| arithmetic mean | 18.0 | 4.1 | 19.7 | 4.0 | 7.4 | 1.8 |

**Table 4.2. WAM Operation Dynamic Instruction Percentages**

Instructions executed within the internal library routines include sixteen percent within the fundamental operations of general unification and fail and thirty percent within built-in predicate routines as shown in table 4.3. The average use of general unification is a low six percent. An exception is the cannibals benchmark that spends significant time unifying two complex data structures. Of the instructions executed for all of unification, only one-eight are in the general unification routine. The high frequency of use of compiled unification code shows the ability of the WAM to specialize the unification process. The average benchmark executes ten percent of its instructions while backtracking and seven percent of its instructions while managing its choice points. To obtain a measure of the total cost of managing the search within each benchmark, we can sum the instructions executed during failure and choice point management. The benchmarks with the highest such measures are routes, grammar, statistics, and compiler that are relatively free of cuts. Rowreduce (containing a considerable number of cuts) and queens use the least percentage of instructions managing their search.

Although the dynamic frequency of use of the built-in predicates of input and output, equivalence check, and classification are twenty-three percent, thirteen percent, and six percent, the percentage of instructions executed for

| Benchmark | Percentage of SIMPLE Instructions | | | | | |
|---|---|---|---|---|---|---|
| | Internals | | Built-In Predicates | | | |
| | Unify | Fail | Math | I/O | Equiv | Class |
| cannibals | 27.7 | 4.6 | 35.3 | 0.3 | 0.0 | 0.0 |
| characters | 5.1 | 9.4 | 8.4 | 0.1 | 0.0 | 3.8 |
| compiler | 2.3 | 14.4 | 21.3 | 2.0 | 0.0 | 0.0 |
| cryptoarithmetic | 7.5 | 10.6 | 27.8 | 0.0 | 0.0 | 4.4 |
| grammar | 3.6 | 18.5 | 0.2 | 5.5 | 0.0 | 0.0 |
| hexconversion | 3.5 | 8.5 | 50.1 | 0.7 | 0.0 | 0.0 |
| primes | 0.3 | 6.4 | 40.1 | 0.5 | 0.0 | 0.0 |
| queens | 1.7 | 2.7 | 49.9 | 0.2 | 0.0 | 0.0 |
| routes | 5.2 | 22.2 | 4.1 | 4.9 | 1.2 | 0.0 |
| rowreduce | 1.7 | 3.9 | 40.3 | 1.6 | 0.0 | 0.1 |
| statistics | 6.1 | 12.5 | 7.3 | 2.4 | 0.0 | 0.0 |
| arithmetic mean | 5.9 | 10.3 | 25.9 | 1.7 | 0.1 | 0.8 |

**Table 4.3. Internal Routines Library Dynamic Instruction Percentages**

their internal routines is rather low. A considerable number of instructions executed, as many as fifty percent for some benchmarks are within the mathematical built-in predicate operations despite few benchmarks that would be considered computationally intensive. A first glance would suggest only the hexconversion and rowreduce to be mathematically oriented. A closer examination of the benchmark code shows perhaps four or five of the benchmarks to include a fair number of arithmetic operations. We have found that arithmetic evaluation and comparison dynamically account for fifty-eight percent of total built-in predicates used in the benchmark executions. The high cost of arithmetic is due not to the frequency of use of mathematical operations as much as the costs incurred from representation and evaluation of arithmetic expressions. Arithmetic is not performed on simple constant values but on structures of arithmetic expressions that must be analyzed and broken down to smaller expressions in the same ways that symbolic expressions are handled. When an arithmetic term is to be evaluated, the term must first be dereferenced and verified to be a structure with arithmetic functor or a numerical constant. If the dereferenced term is a constant, that value is the result. If the term is a structure, its arguments must be evaluated. When the recursive evaluation of the arguments complete, their values are used to calculate the value of the outer-most term. If at any time, a subterm is found to not match the form of an arithmetic expression, backtracking must occur. Arithmetic overflow and underflow must be checked as each calculation is performed. The creation and evaluation of arithmetic expressions and comparisons in this manner incurs very high overhead.

### 4.2. Costs of Fundamental Operations

The contributions of fundamental operations to the number of instructions executed is shown in table 4.4. The most costly of these operations, though its percentage of execution varies greatly among the benchmarks, is evaluate. The grammar benchmark makes practically no use of evaluate while hexconversion and queens execute forty percent of their instructions within evaluate. The average twenty-two percent of instructions executed within this routine accounts for almost all of the time spent executing the arithmetic built-in predicates. The second most costly operation is dereferencing accounting for seventeen percent of the executed instructions for each benchmark. Decdring averages four percent of the instructions executed. The bind and trail operations combine for six and a half percent of instructions executed.

| Benchmark | Percentage of SIMPLE Instructions | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Fundamental Operations | | | | | | | |
| | bind | trail | deref | decdr | fail | detrail | unify | eval |
| cannibals | 1.1 | 1.4 | 17.8 | 9.1 | 3.0 | 1.6 | 27.7 | 29.6 |
| characters | 1.7 | 2.6 | 15.9 | 6.9 | 6.3 | 3.1 | 5.1 | 6.5 |
| compiler | 3.2 | 4.3 | 18.4 | 3.3 | 8.8 | 5.6 | 2.3 | 17.4 |
| cryptoarithmetic | 2.5 | 4.1 | 17.4 | 1.9 | 6.1 | 4.5 | 7.5 | 25.6 |
| grammar | 5.3 | 7.2 | 19.3 | 4.9 | 9.5 | 9.0 | 3.6 | 0.3 |
| hexconversion | 3.2 | 4.6 | 19.0 | 0.2 | 5.3 | 3.2 | 3.5 | 42.9 |
| primes | 1.5 | 1.6 | 13.6 | 4.4 | 5.1 | 1.3 | 0.3 | 32.6 |
| queens | 3.0 | 3.7 | 18.0 | 1.6 | 1.9 | 0.8 | 1.7 | 40.9 |
| routes | 3.7 | 6.1 | 16.8 | 3.0 | 14.0 | 8.2 | 5.2 | 4.0 |
| rowreduce | 2.7 | 3.3 | 16.7 | 2.2 | 2.5 | 1.4 | 1.7 | 34.8 |
| statistics | 2.0 | 3.1 | 13.8 | 5.6 | 8.6 | 3.9 | 6.1 | 6.4 |
| arithmetic mean | 2.7 | 3.8 | 17.0 | 3.9 | 6.5 | 3.9 | 5.9 | 21.9 |

**Table 4.4. Fundamental Operation Dynamic Instruction Percentages**

### 4.3. Enhancements through Mode Analysis and Global Optimization

The performance improvements achieved through the WAM are due to simple compile-time analysis leading to special-case compiled unification code that executes more efficient than a generic unification routine. The concept of modes was introduced by Warren as a way to specify how a procedure argument is to be used as a means to further improve performance through further specialized code [38]. The performance improvements obtained from programmer mode specification encouraged the development of compile-time analysis techniques to automatically determine procedure argument modes that expand beyond simple input/output parameter specification to include more detailed information such as type, dereferencing, and deterministic modes.

The large contributions of compiled unification and dereferencing to static SIMPLE code and the large amount of code that is never executed suggests that a significant amount of static code reduction may be possible by applying compile-time determined mode information to these areas. While the WAM is specialized in comparison to interpretation, it still generates code to allow the full power of Prolog semantics within contexts in which only a subset of situations are realized. The large number of instructions executed in compiled unification and dereferencing and in the evaluation of arithmetic expressions makes these attractive areas to search for dynamic performance improvements through mode analysis and global optimization.

The next few tables show the performance improvements that may be obtained through mode analysis and global optimization in compiled unification code, through elimination of unnecessary dereferencing, and through simplification of the evaluation of arithmetic expressions within the math computation and comparison built-in predicate routines. These results were obtained through a reverse engineering analysis. Dynamic flags and statistic gathering operations were inserted into SIMPLE code by the SIMPLE compiler to identify sections of code responsible for determining the different attributes of data within a program. For example, sections of code were flagged that decided whether a procedure argument was a variable or an instantiated object. Sections of code that verified the type or value of an object were flagged. Through examination of the statistics generated by the SIMPLE simulator, specific sections of flagged decision code were located in which the same decision was made throughout a benchmark's execution. For example, this allowed the identification of locations where it was always the case that an uninstantiated variable was passed into a procedure for a specific argument or where it was always the case that the correct type of object was passed into a procedure. These sections of code signal possible optimizations. An intelligent compiler *might* be able to recognize at compile time that within the context of a specific benchmark a specific decision was predetermined. An optimizing compiler could eliminate the code that performed this decision and code that would be unexecuted along the never taken decision path. The results in this section do not indicate the performance benefits of any particular global analysis technique but represent the potential for improvement for several optimizations through mode analysis for a pure WAM model with our benchmarks. The reader is referred to the many recent publications on specific implementation techniques to achieve automatic mode determination and global optimization for Prolog [88-95].

Table 4.5 and table 4.6 show the static code size reduction and the decrease of instructions executed possible with the application of optimization techniques using a set of mode information within compiled unification code. The first table shows an average static reduction in code size of thirty-nine percent. This represents elimination of forty-nine percent of compiled unification code and fifty-nine percent of unexecuted code. The second table shows an average reduction of instructions executed of nine percent ranging from a low of four percent to a high of fourteen percent. This dynamic instruction reduction represents twenty percent of the instructions executed during

| Benchmark | Percentage of SIMPLE Instructions | | | | |
|---|---|---|---|---|---|
| | Variable / Instantiated | Object Type | Object Value | Structure & List Read / Write | Total |
| cannibals | 5.6 | 0.6 | 0.2 | 51.2 | 57.5 |
| characters | 8.7 | 0.9 | 0.3 | 22.0 | 31.9 |
| compiler | 14.8 | 1.6 | 0.3 | 25.4 | 42.1 |
| cryptoarithmetic | 5.8 | 0.3 | 0.0 | 40.0 | 46.1 |
| grammar | 16.2 | 1.3 | 0.0 | 20.6 | 38.1 |
| hexconversion | 9.1 | 0.7 | 0.0 | 30.1 | 39.9 |
| primes | 8.1 | 0.5 | 0.0 | 19.4 | 28.0 |
| queens | 5.0 | 0.3 | 0.0 | 30.7 | 36.0 |
| routes | 6.9 | 1.0 | 0.0 | 23.7 | 31.7 |
| rowreduce | 6.2 | 0.5 | 0.0 | 33.3 | 40.0 |
| statistics | 10.4 | 0.8 | 0.0 | 28.0 | 39.2 |
| arithmetic mean | 8.8 | 0.8 | 0.1 | 29.5 | 39.1 |

**Table 4.5. Code Size Reductions Using Mode and Type Information**

| Benchmark | Percentage of SIMPLE Instructions | | | | |
|---|---|---|---|---|---|
| | Variable / Instantiated | Object Type | Object Value | Structure & List Read / Write | Total |
| cannibals | 0.6 | 0.4 | 0.1 | 3.5 | 4.6 |
| characters | 2.7 | 2.3 | 0.8 | 8.2 | 14.0 |
| compiler | 3.1 | 1.7 | 0.4 | 3.5 | 8.8 |
| cryptoarithmetic | 1.6 | 0.0 | 0.0 | 5.4 | 7.1 |
| grammar | 4.6 | 2.0 | 0.0 | 5.5 | 12.1 |
| hexconversion | 1.4 | 0.4 | 0.0 | 2.9 | 4.8 |
| primes | 1.9 | 0.8 | 0.0 | 6.4 | 9.1 |
| queens | 1.4 | 0.4 | 0.0 | 5.7 | 7.5 |
| routes | 1.7 | 1.3 | 0.1 | 3.2 | 6.3 |
| rowreduce | 1.7 | 0.7 | 0.0 | 6.3 | 8.7 |
| statistics | 3.0 | 0.3 | 0.0 | 7.4 | 10.6 |
| arithmetic mean | 2.2 | 0.9 | 0.1 | 5.3 | 8.5 |

**Table 4.6. Performance Improvements Using Mode and Type Information**

compiled unification.

Figure 4.1 shows the SIMPLE instruction sequence for a WAM `get_constant` operation. It has a form typical of get and unify operations and is used to illustrate the specific mode techniques used in obtaining the results in tables 4.5 and 4.6. The `get_constant` operation unifies a procedure argument by transforming the argument into a new constant when the argument is originally uninstantiated or by checking that the argument is equivalent to a specific constant value when the argument is originally instantiated. The `get_constant` operation begins by dereferencing its procedure argument (lines 1-8). It determines whether the dereferenced argument is a variable or an instantiated object (lines 9-10). In the case of an instantiated argument, the operation verifies that the argument has an integer tag (lines 11-12) and that it is set to the correct value (lines 13-14). In the case of a variable argument the variable is trailed if necessary (lines 16-27) and given the proper tag (lines 31) and value (lines 30).

The first three columns in tables 4.5 and 4.6 are illustrated by decisions made at lines 10, 12, and 14 of the `get_constant` operation. The variable/instantiated column represents the decision as to whether an argument passed to a procedure is always a variable or never a variable. In the former case, the branch at line 10 is always

```
1    MOV <argument> r1                      get_constant <integer> <argument>
2    AND primary-tag-mask r1 r7             dereference argument
3    BNE r7 variable-tag *+6
4    AND value-mask r1 r7
5    LD  0(r7) r1
6    AND value-mask r1 r8
7    BEQ r7 r8 *+2
8    BRA *-6
9    AND primary-tag-mask r1 r3             variable or instantiated argument ?
10   BEQ r3 variable-tag *+6
11   AND value-mask r1 r3                   instantiated argument mode
12   JNE r3 constant-integer-tag $fail$     integer tag ?
13   AND value-mask r1 r4
14   JNE r4 <integer> $fail$                integer value ?
15   BRA *+18
16   MOV r1 r8                              variable argument mode
17   AND value-mask r8 r8
18   BLE regHB r8 *+5                       trail global stack variable
19   JGT regTR regPDL $trail_overflow$
20   ST  r1 0(regTR)
21   ADD #1 regTR regTR
22   BRA *+6
23   BLE r8 regH *+5                        trail environment variable
24   BLE regB r8 *+4
25   JGT regTR regPDL $trail_overflow$
26   ST  r1 0(regTR)
27   ADD #1 regTR regTR
28   AND value-mask r1 r4
29   AND cdr-tag-mask r1 r1                 save argument cdr bit
30   ORR <integer> r1 r1                    set argument value field
31   ORR constant-integer-tag r1 r1         set argument tag field
32   ST  r1 0(r4)                           store argument
```

**Figure 4.1. SIMPLE Instruction Sequence for the get_constant WAM Operation**

taken. Had the compile known and used this information, lines 9 to 15 could have been eliminated from the static code. Lines 11 to 15 are unexecuted code in this situation. The test in lines 9 and 10 is predetermined and not necessary. Overall, static code decreases by seven instructions. The number of dynamic instructions eliminated is two times the number of executed `get_constant` operations. (That is, two instructions are used in each `get_constant` operation in making this decision.) Along similar lines, if the argument passed to a procedure is always instantiated, then lines 9 to 10 and 15 to 32 can be eliminated. Overall, static code decreases by twenty instructions. The number of dynamic instructions eliminated is two times the number of executed `get_constant` operations. (That is, two instructions are used in each `get_constant` operation in making this decision.)

Column two corresponds to the case when an instantiated procedure variable is always of the expected type. In this case, lines 11 and 12 are unnecessary. Column three corresponds to the case when an instantiated procedure variable is always of the expected value. In this case, lines 13 and 14 are not needed.

The tables show that variable/instantiated determination is most beneficial and reduces our benchmarks by nine percent statically and two percent dynamically. Object type determination reduces static and dynamic instructions slightly less than one percent. Object value determination has a very small benefit.

The fourth column in the tables corresponds to the decision as to whether to execute a unify operation in read or in write mode. In read mode, a unify operation matches its procedure argument with an element of an existing list or structure. In write mode, a unify operation transforms a procedure argument into a new list or structure element. In read mode, a unify operation acts as a get operation. In write mode, a unify operation acts as a put operation. The results in column four correspond to static and dynamic code reductions when this decision can be determined at compile time. These results averaging a thirty percent reduction in static code and a five percent decrease in instructions executed represent the largest potential benefits for optimizations through mode analysis within compiled unification code. This is due to both the length and frequency of the unify operations.

Table 4.7 shows the static and dynamic instruction reductions possible from eliminating dereferencing code through compile-time determined dereferencing modes. Reverse-engineered dereferencing is illustrated in lines 2 through 8 in the `get_constant` operation. Dereferencing analysis is slightly more complex than the mode analysis previously presented. Dereferencing may be eliminated when it is known that an argument contains an immediate reference to an object. This situation is illustrated when the decision in lines 2 and 3 of the `get_constant` operation is always true and when this decision is immediately false when time the operation is begun and the decision in lines 4 to 7 is then true. The former represents a non-variable. The later represents an unbound variable.

Dereferencing may also be eliminated when it is know that an argument is a ground term that has been previously dereferenced. In this case, it may be possible to pass the previously dereferenced value when passing procedure arguments in the compiled code and eliminate the need for dereferencing. This situation is detected by inserting state inspecting instructions within SIMPLE code that track currently active dereferenced variables. When

| Benchmark | Percentage of SIMPLE Instructions | |
|---|---|---|
| | Code Size Reduction | Performance Improvement |
| cannibals | 7.6 | 9.4 |
| characters | 3.4 | 3.6 |
| compiler | 6.4 | 7.5 |
| cryptoarithmetic | 5.3 | 5.8 |
| grammar | 4.3 | 5.3 |
| hexconversion | 7.6 | 8.7 |
| primes | 7.7 | 5.6 |
| queens | 6.8 | 8.5 |
| routes | 3.7 | 3.5 |
| rowreduce | 6.6 | 6.1 |
| statistics | 3.5 | 2.5 |
| arithmetic mean | 5.8 | 6.1 |

**Table 4.7. Performance Improvements Using Dereferencing Mode Information**

arguments are dereferenced and found to be ground terms, they are added to an active dereferenced variable list. When backtracking removes environment variables from the local stack and variables from the global stack by resetting the top of stack addresses of these areas and when detrailing uninstantiates variables, these variables must be removed from the active dereferenced variable list since they either no longer exist in the current state or no longer have the value of their last dereference. Through the active dereferenced variable list, it is possible to determine when a variable has previously been dereferenced and may not need to be dereferenced again if the compiler is able to supply the dereferenced value.

Table 4.7 combines the the static code reductions and dynamic instruction reductions possible through these two types of dereferencing analysis. Static code may be reduced by six percent and six percent of instructions executed may be eliminated through utilization of dereferencing mode information in our benchmarks. These reductions represent thirty-four percent of static dereferencing instructions and thirty-five percent of dynamic referencing instructions.

The high dynamic cost of twenty-two percent for arithmetic expression evaluation is due to Prolog's symbolic representation of arithmetic expressions. Figure 4.2 illustrates an example of code generated for an arithmetic

Prolog Source Code

```
Y1 is Y2 + 3
```

| WAM Operations | Static SIMPLE Instructions |
|---|---|
| `put_structure +/2,X2` | 7 |
| `unify_value Y2` | 71 |
| `unify_integer 3` | 81 |
| `unify_nil` | 43 |
| `put_variable Y1,X1` | 5 |
| `escape is/2` | 4 |
| | 211 |

| Optimized Evaluation of Expression | Static SIMPLE Instructions |
|---|---|
| `extract Y2 value field to X` | 1 |
| `add 3 to X` | 1 |
| `dereference Y1` | 7 |
| `trail Y1` | 10 |
| `create Y1 from X` | 5 |
| | 24 |

**Figure 4.2. Optimized Arithmetic Expression Evaluation**

computation in Prolog. The WAM operations generated for this statement show how Prolog first creates a structure to represent an arithmetic expression and then calls an arithmetic computation built-in predicate routine to disect and evaluate the expression and unify its value with another term. The figure shows nearly two hundred static SIMPLE instructions are needed just to create the arithmetic structure due to the lengthiness of the unify operation SIMPLE sequences. The built-in predicate routine makes use of the evaluate internal library routine. The evaluate routine is a generic routine that must be able to handle any type of object passed to it. It must dereference its arguments, verify that the second argument is a structure that legally represents an arithmetic expression, recursively call itself to evaluate inner subexpressions, perform the arithmetic computation, and unify the result with the first argument that must be either a constant or variable term. Arithmetic in Prolog has major costs first in the creation of arithmetic structures and second in the disection and evaluation of these structures.

When compile-time analysis can determine information about the attributes of objects within an arithmetic expression, much of the overhead of arithmetic can be eliminated. If it is known that the first argument in the Prolog statement of figure 4.2 is an unbound variable and the second argument is a direct reference to a constant term, then this figure illustrates an optimized version of the statement reduced to twenty-four static SIMPLE instructions that eliminates the use of the generic evaluate routine. These are the most common attributes of arguments of statements of this form.

Mode analysis of arithmetic expressions may allow an optimizing compiler to eliminate the need to first create and then disect expressions when it is determined how these expressions will be used. Exposing the details of the evaluate operation by expanding it in-line allows optimizations discussed previously to be applied. Exposing the details of the evaluate operation also allows optimizations of arithmetic calculations commonly employed in other languages. For example, multiplication by a small constant may be performed by a sequence of shifts and adds instead of relying on a generic multiplication routine. In addition, mode analysis of arithmetic expressions can result in reduction of memory usage. Arithmetic expressions are stored on the global stack. Space on the global stack can only be recovered when backtracking occurs. Largely deterministic programs that create many arithmetic expressions may require large global stack space despite the fact that the arithmetic expressions are not needed after evaluation. Much of the global stack space required of the heaviest users of global stack space among the benchmarks (primes, characters, rowreduce, and cannibals) is accountable to arithmetic expressions.

Table 4.8 shows the static and dynamic code reductions possible through optimization of arithmetic computations and comparisons through mode analysis of arithmetic expression arguments. The table shows static code reduction to range from no improvement for benchmarks that are free of arithmetic computation to twenty-six percent for benchmarks that make heavy use of arithmetic computation. The reduction of dynamic instructions takes place because of the elimination of the creation of arithmetic structures and the simplification of the evaluation of arithmetic expressions. Table 4.8 shows the reduction in dynamic instructions for these two cases. On average, six percent of instructions executed are no longer needed due to the reduction of the creation of arithmetic expressions. On average, there is a reduction of seven percent of instructions executed due to the simplification of arithmetic

| Benchmark | Percentage of SIMPLE Instructions | | | |
|---|---|---|---|---|
| | | Performance Improvement | | |
| | Code Size Reduction | expression creation | expression evaluation | Total |
| cannibals | 15.0 | 7.1 | 12.4 | 19.5 |
| characters | 1.7 | 0.5 | 4.7 | 5.2 |
| compiler | 3.9 | 2.3 | 0.0 | 2.3 |
| cryptoarithmetic | 10.6 | 4.1 | 5.2 | 9.3 |
| grammar | 0.0 | 0.0 | 0.0 | 0.0 |
| hexconversion | 18.2 | 11.2 | 14.4 | 25.7 |
| primes | 6.3 | 10.8 | 15.7 | 26.5 |
| queens | 21.9 | 18.3 | 16.1 | 34.5 |
| routes | 8.0 | 0.4 | 0.3 | 0.7 |
| rowreduce | 25.5 | 11.5 | 10.3 | 21.8 |
| statistics | 2.5 | 1.0 | 2.1 | 3.1 |
| arithmetic mean | 10.3 | 6.1 | 7.4 | 13.5 |

**Table 4.8. Performance Improvements Using Optimized Arithmetic Evaluation**

expression evaluation. A combined average of fourteen percent less instructions executed may be achieved through mode analysis of arithmetic expressions. This reduction represents elimination of fifteen percent of compiled unification instructions a reduction of thirty-four percent of instructions executed for arithmetic expression evaluation.

### 4.4. In-line Expansion Enhancements

The tradeoff when choosing between implementation through library routines and implementation through in-line instructions for built-in predicate operations and fundamental operations are shown in table 4.9 and table 4.10. Table 4.9 shows that expanding all built-in predicates to in-line code will increase static code size by fourteen percent but selecting only the smallest built-in predicates for in-line expansion leads to a smaller static increase of four percent with similar performance improvements due to elimination of procedure call overhead of two percent less dynamic instructions.

Table 4.10 shows similar results for the fundamental operations. The in-line expansion of fail, detrail, unify, and evaluate operations lead to static code size increases of forty-one, fifteen, thirty-eight, and sixteen percent, respectively. Trail, dereferencing, and decdring increases code size by twelve, two, and seven percent when they

| Benchmark | Percentage of SIMPLE Instructions | | | |
|---|---|---|---|---|
| | Fastest Execution (All Built-Ins Inline) | | Fast Execution / Small Code Size | |
| | static increase | dynamic decrease | static increase | dynamic decrease |
| cannibals | 30.3 | 2.4 | 4.4 | 2.4 |
| characters | 11.0 | 4.1 | 3.1 | 4.1 |
| compiler | 5.4 | 3.1 | 1.8 | 2.6 |
| cryptoarithmetic | 8.4 | 3.3 | 4.0 | 3.3 |
| grammar | 1.2 | 0.7 | 0.2 | 0.5 |
| hexconversion | 22.1 | 2.8 | 8.8 | 2.7 |
| primes | 14.2 | 3.2 | 3.5 | 3.1 |
| queens | 13.4 | 2.9 | 7.0 | 2.9 |
| routes | 6.5 | 0.5 | 3.3 | 0.4 |
| rowreduce | 22.6 | 2.3 | 9.0 | 2.0 |
| statistics | 14.7 | 0.8 | 0.9 | 0.4 |
| arithmetic mean | 13.6 | 2.4 | 4.2 | 2.2 |

**Table 4.9. Performance Improvements with Optimal Built-In Configuration**

| Benchmark | Percentage of SIMPLE Instructions | | | |
|---|---|---|---|---|
| | Fastest Execution (All Internals Inline) | | Fast Execution / Small Code Size | |
| | static increase | dynamic decrease | static increase | dynamic decrease |
| cannibals | 134.0 | 48.8 | 24.9 | 47.3 |
| characters | 119.4 | 47.5 | 36.0 | 44.8 |
| compiler | 110.9 | 47.8 | 19.7 | 45.1 |
| cryptoarithmetic | 131.3 | 44.4 | 30.6 | 40.0 |
| grammar | 112.3 | 47.6 | 20.6 | 43.9 |
| hexconversion | 139.9 | 43.6 | 39.4 | 41.0 |
| primes | 126.2 | 41.3 | 52.7 | 40.8 |
| queens | 144.8 | 44.3 | 47.3 | 43.6 |
| routes | 113.1 | 45.2 | 19.6 | 40.8 |
| rowreduce | 120.1 | 42.2 | 23.4 | 41.2 |
| statistics | 105.6 | 45.8 | 18.7 | 41.2 |
| arithmetic mean | 123.4 | 45.3 | 30.3 | 42.7 |

**Table 4.10. Performance Improvements with Optimal Fundamental Configuration**

are expanded. The bind operation should always use in-line instructions. It decreases static code size when expanded to in-line instructions. By expanding only the bind, trail, dereference, and decdr operations, we increase static code size by only thirty percent versus more than doubling code size when all fundamental operations are in-line. The forty-three percent decrease in instructions executed using the restricted set of in-line operations is only two percent less than the dynamic instruction decrease with all fundamental operations expanded in-line.

As illustrated with the evaluate routine, in-line expansion of internal library routines may lead to further optimizations by exposing the details of their operation to special-case analysis and global optimization.

### 4.5. Summary of WAM-Level Enhancements

Optimizations that make use of procedure argument mode information at compile time within compiled unification operations reduce our benchmarks' size by forty percent and decrease the number of instructions executed by nine percent. The most useful mode information is identification of read and write mode within unify operations for structure and list elements. Predetermining whether an argument is a variable or an instantiated object at compile time is beneficial as well. A smaller benefit is obtained by determining an argument's type or value at compile time.

Knowledge of dereferencing characteristics reduces the benchmarks' static code size by six percent and reduces instructions executed within the simulations by six percent through eliminating dereferencing operations for objects that are directly referenced and by eliminating dereference operations when an object has previously been dereferenced and this dereferenced value may be made substituted for the non-dereferenced value. Optimization of arithmetic expression representation and arithmetic expression evaluation eliminates six percent of the benchmarks' executed instructions that create arithmetic expressions and eliminates seven percent of the benchmarks' executed instructions that evaluate these arithmetic structures. Arithmetic expression optimization decreases the static code size of the benchmarks by ten percent.

# CHAPTER 5

# Architectural Benchmark Characteristics

We first present architectural operation profiles including a high-level profile and an opcode/operand level profile. We then propose several enhancements at the architectural level and analyze the performance improvements attained with the implementation of these enhancements.

## 5.1. High-Level Architectural Operation Profile

Six groups of high-level architectural operations in decreasing order of their contributions to dynamic instruction count are tag-handling operations, accesses to different areas of memory, addressing computations, procedure return and calls, arithmetic calculations, and system operations. The percentage of instructions executed within these groups is fairly consistent across the benchmarks. Figure 5.1 shows the average of the benchmark dynamic instruction count percentages for the operations contained within each group.

Five basic operations are involved in handling tagged objects. Tag checking is the operation of verifying that a selected group of bits within the tag field of an object has a specific pattern. Tag extraction is the operation of isolating the tag field (or a portion of the tag field) from an object. Value and address extraction operations isolate the value field from an object. Tag-value combining takes a tag pattern and a value to form a single object.

Tag handling operations use logical and shift instructions. As an example, consider the operations involved in checking that an object is an integer of some particular value as illustrated in figure 5.2. The tag bits of the object in question must be extracted by `AND`'ing the object with a pattern that selects those bits that determine its primary type (tag extraction). These bits must be compared to the tag pattern that identifies an integer (tag checking). The value field of that object is then extracted by `AND`'ing the object with a pattern that removes the objects's value bits (value extraction). (In cases where the value will be used in an arithmetic computation, the value is extracted by shifting the object's bits first left and then right to obtain a properly sign-extended value.) The extracted value is

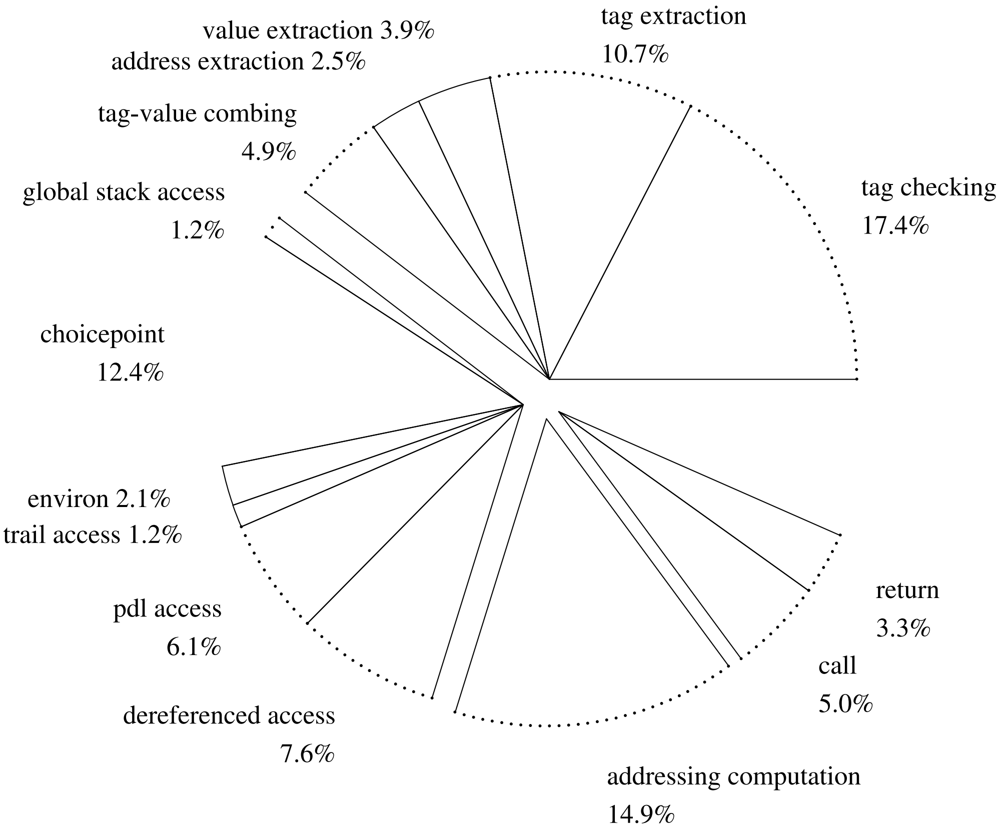


**Figure 5.1. Architectural Operations Dynamic Instruction Percentages**

Verifing that an <object> has an integer-tag and a particular integer-value:

```
AND primary-tag-mask <object> <temporary>     tag extraction
BNE <temporary> integer-tag fail              tag checking
AND value-mask <object> <temporary>           value extraction
BNE <temporary> integer-value fail

                    - or -

AND value-mask integer-value <temporary>      tag-value combining
AND tag-mask integer-tag <tag-temporary>
OR <tag-temporary> <temporary> <temporary>
BNE <temporary> <object> fail
```

**Figure 5.2. Tag-Handling Operation SIMPLE Instruction Sequences**

compared with the particular integer value that is to be checked. Alternatively, we could create a new object having an integer tag and the integer value that is to be checked by OR'ing these subparts together (tag-value combining) and then compare the created object with the object in question in a single comparison.

Tag-handling operations account for almost forty percent of the instructions executed in our benchmarks. Tag checking is the most significant (seventeen percent) of these operations. Tag extraction is the next most significant (eleven percent). Extracting the value field of an object accounts for six percent of the instructions executed. Combining tag and value fields into new objects contributes five percent to executed instructions.

The percentage of instructions executed in accessing all memory areas averages thirty-one percent. Figure 5.1 shows how memory accesses for the different memory areas are divided as determined by the addressing mode of the instruction. We can determine the memory area that will be referenced by the register in an address specification for seventy-five percent of memory references. For example, registers assigned to the choice point, top of trail, and top of global stack address pointers always reference a location within the area with which they are associated. The remaining twenty-five percent of the memory references in which the register in the address computation does not determine the area that will be accessed occur during the dereferencing of variables and when accessing elements of lists and structures. The dereferenced location of an argument register variable or the location of a list or structure element may be within the global stack or it may be within an environment in the local stack. Choice point access accounts for twelve percent of instructions executed. Accessing the push-down list causes six percent of instructions executed. Accessing the trail stack accounts for one percent of dynamic instructions. The combined accesses to environments and the global stack including dereferenced variables sum to eleven percent of executed instructions.

Addressing operations include incrementing and decrementing address pointers when pushing and popping items on the stacks, comparing addresses to determine whether an object resides within a particular area, checking that memory areas do not exceed their static bounds, and other address calculations. Addressing accounts for fifteen percent of instructions executed. One-fifth of this value is due to stack-overflow checking.

Procedure call operations include initial calls to a predicate, subsequent calls to alternative clauses within a predicate, and calls to built-in routines. Executing these three types of calls cost one-half of one percent, nine-tenth of one percent, and four percent of the instructions executed, respectively. Returns from escapes are three percent of the instructions executed. The percentage of return operations from predicate calls is almost unnoticeable. The difference between the cost of calls and the cost of returns to and from predicates occurs because some calls in Prolog never return in the traditional sense. Backtracking can be viewed as a back-door method of ending a procedure's execution. When backtracking, execution of the active procedure is abruptly ended; the return code for the procedure is not executed. The backtracking routine determines a new path of execution from the choice point alternate clause address. A procedure return may be viewed as forward progress in resolving goals to be proven. The existence of non-determinism causes some procedures to begin their execution, but halt prematurely before a "normal" termination.

The last two high-level operation groups, actual arithmetic for mathematical operations specified in the source code and system operations such as reading and writing values, are negligible portions of the executed instructions.

The dynamic distribution of the six groups of operations exhibits some of the distinguishing features and characteristics of Prolog. Dynamic tag handling, being unique to symbolic languages such as Lisp and Prolog, does not occur in procedural languages. Steenkiste and Hennessy report Lisp to use twenty-one percent of its dynamic instructions for tag-handling [60]. Both Prolog and Lisp have a strong need to break apart objects, examine their types, and glue objects together. The larger percentage of tag-handling operations in Prolog is created by the manipulation of tagged data during the large portion of execution involved in unification. The large number of instructions for accessing the choice point stack and the trail stack, especially in comparison to other memory accesses, shows the expense incurred in Prolog to support non-determinism. The support of procedures is less important in Prolog than in Lisp and procedural languages. The percentage of procedure call and return instructions in Lisp was found to be twenty percent by Steenkiste and would be higher in a typical procedural benchmark. Although Prolog programs are built of many small facts and rules, the compilation from Prolog source code to a register-oriented instruction set, such as SIMPLE, and specifically, the large static expansions in translating from Prolog to the WAM and the WAM to a non-WAM machine language makes it more difficult to write procedures that translate to small compact machine language routines in Prolog than in other languages. The major component of a typical procedural language benchmark is the execution of instructions for arithmetic computation. Steenkiste finds Lisp to include six percent of dynamic instructions for arithmetic computation. The lack of arithmetic operations in Prolog reflects both the types of applications for which Prolog is most popular and the large effort needed elsewhere for symbolic processing and support of non-determinism.

## 5.2. Low-Level Architectural Operation Profile

### 5.2.1. Architectural Opcode Profile

Opcode frequency distributions are averaged over the benchmarks and shown in table 5.1. The table shows that register-to-register instructions (move, math, logical and shift) account for forty-three percent of static code and forty-one percent of dynamic instructions. Comparison instructions (branch, jump) are thirty-four and twenty-nine

| | Percentage of SIMPLE Instructions | |
|---|---|---|
| Operation | Static | Dynamic |
| Move | 5.4 | 5.8 |
| Math | 8.9 | 10.9 |
| Logical and Shift | 29.0 | 23.9 |
| Branch | 23.9 | 18.6 |
| Jump | 10.6 | 10.0 |
| Load | 10.1 | 18.8 |
| Store | 10.7 | 11.8 |

**Table 5.1. Opcode Frequency Distributions**

percent of the static and dynamic code. Load and store instructions account for twenty-one percent of the static code and thirty-one percent of the dynamic code. These frequencies are consistent across the benchmarks.

Move instructions contribute five percent to the static code and six percent to the dynamic code. For the most part, move instructions are used to transfer values to temporary registers that can be altered while preserving the original values. These include moves to temporary registers in fundamental operations, moves to temporary registers during tag-handling, and copying values to temporary registers during addressing calculations. Move instructions are used to transfer values between argument registers before a procedure call. Some move instructions are explicitly needed to implement transfers with the semantics of WAM operations.

Integer addition and subtraction accounts for nine percent of the static code and eleven percent of executed instructions. These instructions predominantly serve address calculation needs. Benchmarks that are more computationally extensive do not necessarily have higher math instruction frequencies than benchmarks that are less computationally intensive. The percentage of math instructions assisting in pushing and popping values on the global stack, trail stack, and push-down list as well as other addressing calculations account for eighty-nine percent of the math instruction use. These uses are distributed by the amount of activity in the different stack areas in the individual benchmarks. On average, push and pops to the push-down list cause sixty percent of all additions and subtractions. Addressing in the global stack and trail stack each add ten percent to dynamic math instruction use. Ninety-six percent of the arithmetic instructions use an immediate value as their first operand.

Logical and shift instructions have the highest frequencies of twenty-nine and twenty-four percent of static and dynamic instruction use. These instructions are used for tag and value field extraction, tag and value field combining, and sign extensions for the value fields. Nearly all logical and shift instructions manipulate the contents of a register with an immediate value operand.

Branch instructions comprise twenty-four percent of the static code and nineteen percent of the instructions executed. Jump instructions comprise eleven percent of the static code and ten percent of dynamic instruction use. Generally, branches are used within a procedure and jumps transfer control between procedures and to the internal library routines. Branch instructions are used to make comparisons between the tag of a value within a register and an immediate tag value, comparisons between two value fields or two addresses, and to branch unconditionally around code within a procedure. Jump instructions are also used to compare tags, values, and addresses. Procedure calls and procedure returns are implemented through unconditional jumps. There are twice as many branch instructions executed as jump instructions. Approximately ten percent of branch instructions and eleven percent of jump instructions are unconditional. The majority of conditional branches are taken; however, only ten percent of the few branches that have backward-target addresses are taken which shows that loops implemented through branch instructions are infrequent and when they do occur are not often taken. Eighty-seven percent of conditional branches check if two values are equal or not equal. The majority of conditional jumps are not taken. Many of these instructions are tests that initiate backtracking when successful. Most often the test condition fails giving partial satisfaction of a unification to be performed. Fifty percent of conditional jumps check if two values are equal or not equal. The other half of conditional jumps check for conditions of inequality.

Load instructions, including instructions that transfer a value from memory to a register as well as instructions that transfer a constant to a register account for ten percent of the static code and nineteen percent of the dynamic code. Store instructions, including instructions that transfer a constant to memory and instructions that transfer a register's contents to memory occupy eleven percent of the static code and twelve percent of the dynamic code. Load instructions are used to set argument register values from an environment variable, or from the global stack, or to an immediate value. They are used during backtracking when resetting register values from the choice point and upon the return of a procedure to reset environment registers. During dereferencing and decdring, objects are loaded from memory. During detrailing, the addresses for locations that are to be uninstantiated are loaded from the trail stack. Objects are loaded into registers when accessing the push-down list. Store instructions create elements on the global stack and elements within environments. Values are moved to memory when creating a choice point, when trailing and detrailing variables, and when there is a need to store values on the push-down list.

Most often, the register used in memory addressing indicates the WAM area of the addressed memory object. This is always the case with objects within the choice point stack, trail stack, and push-down list, each of which has a specific register assigned to access its contents. Access to the trail stack and push-down list follow a strict last-in, first-out ordering. The choice point stack most often acts as a last-in,first-out stack, but includes other references within the top-most choice point that contains sixteen elements. References to the global stack occur at the top of the global stack when creating a new list or structure element, but may occur anywhere within its contents when accessing an existing object. References to environment variables within the local stack occur in the top-most environment (with greatest size of fourteen words in our benchmarks) when using the environment pointer, but may occur within any environment when accessing existing data. It is during the dereferencing of objects, when following list and structure pointers, and when accessing elements of a list or structure that internal references are generated to the global stack and local stack.

Figure 5.3. compares our dynamic opcode distribution with similar measurements for Pascal and Lisp as reported by Steenkiste and Hennessy [59]. This figure shows that instruction frequency differences between Prolog and Pascal magnify most differences between symbolic processing and numerical processing when they are compared to the instruction frequency differences of Lisp and Pascal.

All three languages use the same percentages of computational instructions (move, arithmetic, logical and shift). The most striking difference among computational instructions among the three languages is how their use differs between arithmetic and logical and shift operations. The high use of logical and shift instructions in Prolog is

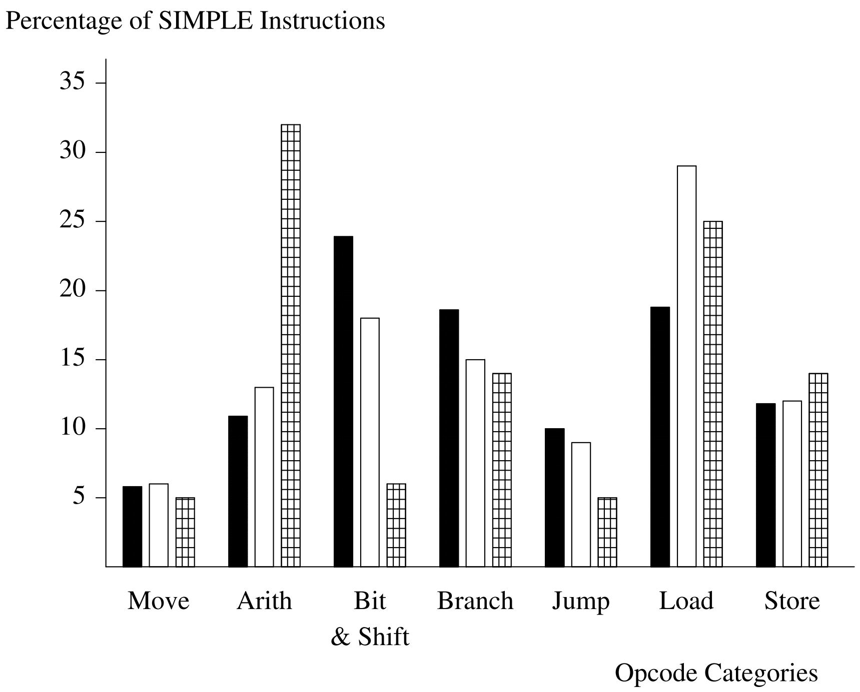


**Figure 5.3. Prolog (black), Lisp (white), and Pascal (checked) Instruction Profiles**

caused by the tagged representation of objects in the WAM model. The low use of arithmetic instructions is due to Prolog's preference towards symbolic processing applications. Most of the activity within the WAM model and within the applications most suited for its use is not heavily computational. As important, because Prolog represents arithmetic expressions in the same way in which it represents other objects, a computationally intensive application will still spend significant time creating and manipulating the tagged data structures of arithmetic expressions when performing computations.

Our instruction profile shows higher frequencies for both branch and jump instruction use in Prolog than in Lisp and Pascal. The high frequency of branch and jump instructions are due to their use in unification operations. There is a frequent need to compare and match the corresponding fields of different objects and then act accordingly.

The operations in which Prolog has lower frequencies of use than Pascal and Lisp are loads and stores. Although one may think that this decreased percentage of memory references will reduce the concern of memory access time, these concerns are not lessened. Because Prolog's data representation (more specifically, the heavy use of pointers) reduces the spatial locality of memory references, efficient memory design is very important to an efficient Prolog system.

### 5.2.2. Architectural Operand Profile

Operands may reside in registers, or in the different memory areas, or may be immediate values specified in the code. Table 5.2 shows how these operand types are dynamically distributed in the benchmarks. On average, registers, immediate values, and memory operands account for fifty-six, thirty-three, and eleven percent of operand references.

Table 5.3 shows the distribution of reads and writes within the different memory areas. The local stack is most heavily accessed, followed by the push-down list, and the global stack. The different benchmarks vary quite a bit with their use of the global stack, local stack, and push-down list. The characters, compiler, grammar, routes, and statistics benchmarks heavily access the local stack because they frequently create, modify, and read choice points. The push-down list is accessed frequently by the cannibals, hexconversion, and queens benchmarks. In cannibals, this is due to the high use of general unification. The hexconversion and queens benchmarks make heavy use of the math built-in predicate. These routines make use of the push-down list for storing their recursive data

| Benchmark | Percentage of Operand References | | |
|---|---|---|---|
| | Register References | Memory References | Immediate References |
| cannibals | 55.09 | 9.46 | 35.44 |
| characters | 55.44 | 10.35 | 34.20 |
| compiler | 55.75 | 11.43 | 32.80 |
| crytoarithmetic | 56.87 | 11.17 | 31.94 |
| grammar | 56.56 | 10.20 | 33.23 |
| hexconversion | 57.22 | 10.72 | 32.04 |
| primes | 56.87 | 10.66 | 32.45 |
| queens | 57.30 | 9.01 | 33.67 |
| routes | 55.60 | 13.07 | 31.32 |
| rowreduce | 56.89 | 9.57 | 33.52 |
| statistics | 55.24 | 11.72 | 33.03 |
| arithmetic mean | 56.26 | 10.67 | 33.06 |

**Table 5.2. Dynamic Operand Reference Distribution**

| Benchmark | Percentage of Memory References | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Atom Space | Heap | | Local Stack | | Trail Stack | | PDL | |
| | rd | rd | wr | rd | wr | rd | wr | rd | wr |
| cannibals | 0.1 | 15.8 | 6.0 | 19.4 | 15.1 | 0.6 | 0.6 | 20.2 | 22.2 |
| characters | 0.0 | 10.1 | 5.1 | 32.8 | 32.3 | 1.0 | 1.3 | 8.0 | 9.5 |
| compiler | 0.4 | 8.0 | 5.7 | 38.5 | 22.1 | 1.9 | 1.9 | 10.5 | 11.1 |
| cryptoarithmetic | 0.0 | 10.3 | 6.9 | 28.6 | 21.1 | 1.7 | 1.7 | 14.8 | 14.9 |
| grammar | 2.3 | 10.3 | 13.4 | 42.4 | 19.7 | 4.1 | 4.1 | 1.8 | 1.8 |
| hexconversion | 0.1 | 10.8 | 5.6 | 25.8 | 14.1 | 1.1 | 1.1 | 21.0 | 20.4 |
| primes | 0.2 | 12.2 | 5.5 | 26.1 | 21.9 | 0.0 | 0.2 | 17.1 | 16.6 |
| queens | 0.0 | 13.1 | 8.9 | 17.0 | 15.5 | 0.2 | 0.2 | 23.3 | 21.8 |
| routes | 1.4 | 4.3 | 3.1 | 52.7 | 26.3 | 2.3 | 2.3 | 3.6 | 4.0 |
| rowreduce | 0.2 | 13.6 | 9.4 | 19.4 | 21.2 | 0.5 | 0.5 | 18.1 | 17.1 |
| statistics | 0.7 | 7.2 | 4.1 | 39.4 | 32.4 | 1.0 | 1.0 | 6.5 | 7.8 |
| arithmetic mean | 0.5 | 10.5 | 6.7 | 31.1 | 22.0 | 1.3 | 1.4 | 13.2 | 13.4 |

**Table 5.3. Dynamic Memory Reference Distribution**

structures. The cannibals, grammar, and rowreduce benchmarks access the global stack very often.

An interesting observation from table 4.3 is that more writes than reads take place in the trail stack and push-down list. Elements in both of these stacks are generally written once (pushed) and read once (popped). The push-down list is used as a means of transfering data between procedure calls. When unification fails, the push-down list is completely cleared and existing items on the list are never read. The trail stack is used to record variables that need to be uninstantiated. When the inital goal of a program completes, most likely, the trail stack contains some set of addresses that have not be read.

Immediate values may be integer constants, atoms, absolute code-space addresses, or code-space address offsets. Data-space immediate values may be grouped into tag constants, small constants with values of absolute value less than 127, integer constants that point to stack boundaries, other large integer values, and atoms. Fourteen distinct tag patterns are used to implement the WAM representation of objects. The atom constant value field points to a symbol table and must be able to distinguish between all elements in the table. The size of the stack boundary values is dependent on the size of the different memory areas. Counting both tag and small constants, over ninety percent of the constants used can be represented by eight bits or less.

Offsets can be grouped into large negative offsets with values less than -128, small negative offsets with values greater than or equal to -128, small positive offsets with values less than or equal to +127, and large positive offsets with a value greater than +127. Ninety percent of all offset values are small positive values used to branch within sections of code within WAM operation sequences. Eight percent are small negative values, mostly used in loops in the dereferencing and decdring operations and others within the internal library routines. All but two percent of offsets may be represented in eight bits.

### 5.3. Enhancements through Tag-Handling Architectural Support

The high dynamic cost of tag-handling operations make them a suitable target for architectural enhancement. Tag-handling is expensive because our base architecture treats each object as a single entity. Examining or manipulating either of the tag or value fields requires extracting that field. Creation of new objects requires a combining operation after the tag and value fields are created independently. The architectural characteristic of single-entity objects contrasts with the WAM's data representation of independent tag and value fields and results in Prolog's large frequency of logical and shift operations. Many of these instructions will be eliminated through the following enhancements.

Table 5.4 and table 5.5 display the static and dynamic benefits of the tag-handling architectural enhancements. The tables show the reduction in static and dynamic instruction use for each enhancement as a percentage of SIMPLE instructions using our base architecture simulations. Overall, our results show that the combined effects of these tag-handling enhancements can lead to a twenty-three percent reduction in static code size and a twenty percent reduction of instructions executed. These enhancements reduce the static code generated for tag handling by sixty percent and reduce the dynamic instructions executed during tag handling by fifty percent. Seventy-five percent of logical and shift instructions are eliminated from the static code and eighty-five percent of the dynamic logical and shift instructions are no longer necessary.

Our first architectural enhancement to assist in tag operations is to force the architecture to recognize that each data object consists of a tag field and a value field and to treat each field appropriately for different operations in the instruction set. As an example, an add instruction should only add the value fields of its two source operands and store the result of its operation in the value field of its destination operand. A second example would be in the use of

| Benchmark | Percentage of SIMPLE Instructions | | | |
|---|---|---|---|---|
| | Tagged Architecture | Masked Tag Comparison | Masked Tag Insertion | Total |
| cannibals | 12.2 | 9.1 | 1.5 | 22.8 |
| characters | 11.9 | 8.2 | 1.3 | 21.4 |
| compiler | 12.8 | 8.9 | 1.7 | 23.4 |
| cryptoarithmetic | 12.2 | 8.6 | 1.3 | 22.1 |
| grammar | 13.3 | 9.1 | 1.9 | 24.3 |
| hexconversion | 12.7 | 8.1 | 1.4 | 22.2 |
| primes | 11.5 | 8.3 | 1.2 | 21.0 |
| queens | 12.1 | 8.2 | 1.1 | 21.4 |
| routes | 13.8 | 8.6 | 1.8 | 24.2 |
| rowreduce | 11.4 | 8.2 | 1.2 | 20.8 |
| statistics | 14.2 | 8.7 | 2.0 | 24.9 |
| arithmetic mean | 12.6 | 8.5 | 1.5 | 22.6 |

**Table 5.4. Code Size Reductions Using Tag Operation Support**

| Benchmark | Percentage of SIMPLE Instructions | | | | |
|---|---|---|---|---|---|
| | Tagged Architecture | Masked Tag Comparison | Masked Tag Insertion | Concurrent Tag Operations | Total |
| cannibals | 6.7 | 12.5 | 0.5 | 2.1 | 21.8 |
| characters | 4.1 | 12.8 | 0.3 | 1.4 | 18.6 |
| compiler | 7.7 | 9.8 | 0.7 | 2.0 | 20.2 |
| cryptoarithmetic | 6.7 | 9.6 | 0.8 | 2.1 | 19.2 |
| grammar | 9.2 | 10.3 | 1.0 | 3.0 | 23.5 |
| hexconversion | 8.4 | 8.2 | 1.1 | 2.7 | 20.4 |
| primes | 5.5 | 10.2 | 0.6 | 2.0 | 18.3 |
| queens | 7.1 | 9.8 | 1.1 | 2.9 | 20.9 |
| routes | 7.0 | 9.5 | 0.8 | 2.2 | 19.5 |
| rowreduce | 7.0 | 10.0 | 1.0 | 2.6 | 20.6 |
| statistics | 4.2 | 11.7 | 0.4 | 1.8 | 18.1 |
| arithmetic mean | 6.7 | 10.4 | 0.8 | 2.3 | 20.1 |

**Table 5.5. Performance Improvements Using Tag Operation Support**

a register in an address computation. Only the value field of the register should take part in computing an effective memory address. The performance of each of these examples is improved by eliminating the need to extract a field from an object prior to operating with the field's value. Certain operations, (for example, moves, bit masking, and comparisons), are appropriate for both tag fields and value fields. For these operations, instructions can be defined to include cases to apply the operation to the tag fields of its operands, the value fields of its operands, or possibly both fields of its operands, through different operation codes. The enhancement of recognizing tagged objects within the architecture reduces the code size of our benchmarks by thirteen percent and reduces the number of dynamic instructions by seven percent.

The next enhancements result from the addition of three new instructions. The tag bits of an object are used for several purposes: to identify the type of an object, to identify the subtype of an object, for cdr-coding, for garbage collection, and for specifying other information. Most tag checking operations compare a selected subset of the bits of an objects' tag to a specific pattern. The isolation of just the tag field of an object does not avoid the masking that must be included within many tag-checking operations. A combined mask and compare instruction would reduce operations of this type from two instructions to a single instruction. Two new instructions could be defined that use an immediate mask value to select the appropriate tag bits of a register's contents and compare those bits to an immediate pattern value. One instruction would test for equality. The other would test for inequality. The outcome of the comparison would determine if execution should continue with the following instruction or branch based on the value of the instruction's target-destination field. Addition of masked tag-comparison instructions reduces the number of static instructions by nine percent and the number of dynamic instructions by ten percent.

A third new instruction that would be useful would perform a combined mask and move operation. This instruction would assist in operations with a need to insert a pattern of tag bits into a subset of the bit positions of an object's tag field. The new instruction would be defined to use an immediate mask value to select the appropriate tag bits within a register to alter and set those bits to the corresponding bits of an immediate pattern. The addition of the masked tag insertion leads to a small performance gain by reducing static instructions by two percent and reducing dynamic instructions by one percent.

A final tag-handling enhancement would be to allow operations on tag fields and operations on value fields to proceed in parallel since these two types of objects are now distinct and will have no data dependencies between them. The overlapping of tag operations would not reduce the static size of programs but would result in a small overlapping of two percent of dynamic instructions.

One small, but additional benefit results in a tagged architecture when math overflow is checked through hardware. Table 5.6, shows one-half of one percent of dynamic instructions could be eliminated if software did not have to check that the result of an arithmetic computation overflowed the bits reserved for an object's value field as is the case for a non-tagged architecture.

### 5.4. Enhancements through Stack Support

The most direct way to support the stack-like memory areas of a WAM implementation through enhancement of an instruction set is through the inclusion of push and pop instructions. The pure last-in, first-out stack access of the trail stack and the push-down list and the frequent top-of-stack access when adding items to choice points and the global stack make push and pop instructions beneficial. Table 5.7 shows that the addition of push and pop instructions leads to a decrease in static code size of seven percent and a performance improvement by reducing the number of executed instructions by eight percent in comparison to simulations with our base architecture. These

| Benchmark | Percentage of SIMPLE Instructions Performance Improvement |
|---|---|
| cannibals | 0.6 |
| characters | 0.0 |
| compiler | 0.1 |
| cryptoarithmetic | 0.7 |
| grammar | 0.0 |
| hexconversion | 0.9 |
| primes | 0.6 |
| queens | 1.1 |
| routes | 0.1 |
| rowreduce | 0.9 |
| statistics | 0.1 |
| arithmetic mean | 0.5 |

**Table 5.6. Performance Improvements Using Math Overflow Support**

| Benchmark | Percentage of SIMPLE Instructions | |
|---|---|---|
| | Code Size Reduction | Performance Improvement |
| cannibals | 7.7 | 11.0 |
| characters | 7.6 | 6.8 |
| compiler | 7.1 | 7.7 |
| cryptoarithmetic | 7.8 | 9.5 |
| grammar | 6.9 | 5.3 |
| hexconversion | 7.5 | 11.2 |
| primes | 7.8 | 9.7 |
| queens | 8.0 | 10.6 |
| routes | 6.9 | 5.0 |
| rowreduce | 7.5 | 9.3 |
| statistics | 6.6 | 6.5 |
| arithmetic mean | 7.4 | 8.4 |

**Table 5.7. Performance Improvements Using Push / Pop Operations**

improvements eliminate eighty-three percent of static math instructions and seventy-seven percent of dynamic math instructions. They account for forty percent of static addressing operation instructions and fifty-six percent of dynamic addressing operation instructions.

Because the four WAM data-space areas are dynamic in size, a Prolog system must check that areas do not cross fixed static boundaries or must insure that an area does not grow into another area's allocated space. Several characteristics of Prolog systems make stack-overflow checking more important and more difficult in Prolog than in languages which do not have implicit data allocation. While it is difficult to estimate a program's minimum space requirements without executing the program, Prolog implementations have generally neglected to include support for dynamic stack expansion. The extra stacks unique to a Prolog implementation cause extra fixed static boundaries. It is not possible for all stacks to be organized to grow towards each other. Thus, many Prolog systems require the user to specify the minimum space requirements for each of the WAM areas. Most Prolog areas grow an object at a time which may force frequent stack-overflow checking. Prolog includes no instructions to dynamically allocate a block of memory.

SIMPLE accomplishes stack-overflow checking through explicit instructions scattered throughout a program's code which check each stack expansion against currently allocated boundaries. Table 5.8 shows that if it were

| Benchmark | Percentage of SIMPLE Instructions | |
|---|---|---|
| | Code Size Reduction | Performance Improvement |
| cannibals | 5.9 | 2.9 |
| characters | 5.6 | 2.9 |
| compiler | 5.8 | 2.6 |
| cryptoarithmetic | 5.7 | 3.6 |
| grammar | 5.6 | 2.8 |
| hexconversion | 5.3 | 3.2 |
| primes | 4.7 | 3.0 |
| queens | 5.4 | 3.6 |
| routes | 5.8 | 2.1 |
| rowreduce | 6.0 | 3.4 |
| statistics | 5.8 | 2.4 |
| arithmetic mean | 5.6 | 3.0 |

**Table 5.8. Performance Improvements Using Stack Overflow Support**

possible to eliminate these checks, static code size would be reduced by six percent and the number of instructions executed would decrease by three percent.

Alternative implementations for stack-overflow checking can include a combination of software and hardware or hardware-exclusive solutions. An example of the former is the multi-stack management scheme of the PSI machines that eliminates the need for stack-overflow checking by allowing stacks to grow to any size [80, 42]. The memory system of the PSI is similar to traditional virtual memory systems. Each stack area is implemented as a virtually-addressed memory segment built from a pool of fixed-size pages. Hardware support for the PSI includes page-map base registers and fast page-map memory to assist a complex address translation mechanism and detects when a stack has out grown its allocated size. Software allows the allocation of additional pages to stack areas when necessary and the deallocation of pages to a free-page pool when space must be reclaimed from stacks that have decreased in size.

Hardware stack-overflow solutions include the use of stack-boundary registers and page-protection mechanism schemes. In the WAM, when an area is expanded because a new object is created at the top of its stack, it is always the case that the WAM address-pointer register that references the new object location is associated with that particular area. Area-specific address-pointer registers may have accompanying boundary registers that are loaded at the start of a program's execution with the largest (or smallest) allowable address for the stack with which the address pointer is associated. When an area-specific register is used in addressing, hardware can verify that its contents do not exceed the bound specified in its boundary register. Alternatively, each area-specific register may be granted the privilege to write only to specific pages. Each page would include a tag specifying those area-specific registers that were privileged to write to the page. In a page-protection scheme, hardware would detect when an attempt was made to write to a page with an area-specific register that did not have write access to the page. Both stack-boundary and page-protection stack-overflow checking schemes can be adapted to take advantage of an organization in which two stack areas grow towards one another.

### 5.5. Summary of Architectural-Level Enhancements

Architectural support for tag-handling operations through design and use of an architecture which recognizes the existence of tag and value fields within an object leads to a twelve percent decrease in static code size and a seven percent decrease in executed instructions in the benchmark simulations. The addition of orthogonal tag instructions reduces code size by an additional nine percent and reduces dynamic instructions by eleven percent. Allowing instructions which operate solely on tag fields to execute in parallel with instructions which operate solely on value fields results in a two percent overlapping of instructions. The introduction of push and pop instructions decreases the benchmarks' code size by seven percent and the number of instructions executed by eight percent.

# CHAPTER 6

## Instruction-Level Parallelism

Pipelining is an effective technique to increase parallelism at the instruction level [105]. Ideally, pipelining offers a speedup equal to the number of pipeline stages introduced in an architecture's implementation. In practice, instruction streams include both control dependencies and data dependencies that limit actual performance because these dependencies reduce the amount of parallelism that can be realized [106]. A control dependency occurs in the case of every conditional branch or conditional jump because the instruction that will be executed following the branch or jump is determined by the outcome of the branch or jump decision. A data dependency occurs when one instruction establishes a result that is used as input in a subsequent instruction. Data dependencies may arise through computational instructions and through load instructions. Both control and data dependencies result in a stall of a pipelined execution until all needed information becomes available for a following instruction to be executed. The empty cycles during a pipeline stall act to reduce pipelined performance from ideal levels.

Superscalar and very long instruction word (VLIW) machines exploit instruction-level parallelism by initiating multiple operations simultaneously [107, 108]. Superscalar machines issue more than one instruction per clock cycle when their hardware can guarrentee the independence of a series of instructions. In a VLIW machine, the compiler creates a package of operations that can be simultaneously issued without additional hardware assistance. Ideally, superscalar and VLIW machines offer a speedup equal to the number of concurrent operations issues allowable by the hardware's instruction and execution units. In practice, dependencies again restrict the parallelism that can be attained through static compile-time analysis or through dynamic analysis during a program's execution. The inability after compile-time analysis and runtime analysis to achieve the maximal allowable operations to be issued per cycle reduces multiple-operation issue performance from ideal levels.

A basic block is a sequence of instructions following a branch or jump instruction up to and including the very next branch or jump instruction. The division of a program into basic blocks identifies and separates the control dependencies of a program. Basic blocks are important when considering instruction-level parallelism. In a pipelined machine, a basic block represents a segment of code whose instructions are sequentially and continuously fed into the instruction stream dependent upon the data dependencies within these instructions and the delays that must take place between them. In a multiple operation issue machine, the simultaneous scheduling of instructions within a basic block reduces compile-time analysis to data dependency analysis for which algorithms are well understood. The scheduling of instructions across basic blocks forces the use of experimental, more costly, and more complex compilation schemes.

The next three figures show the frequency of branches and jumps in the benchmarks by looking at how they are distributed throughout the code. Figures 6.1 and 6.2 show the frequencies of static and dynamic basic block size among the benchmarks. The overall average basic block size for the collective benchmark static code is three instructions. The overall average dynamic basic block size for the collective benchmarks is four instructions. The

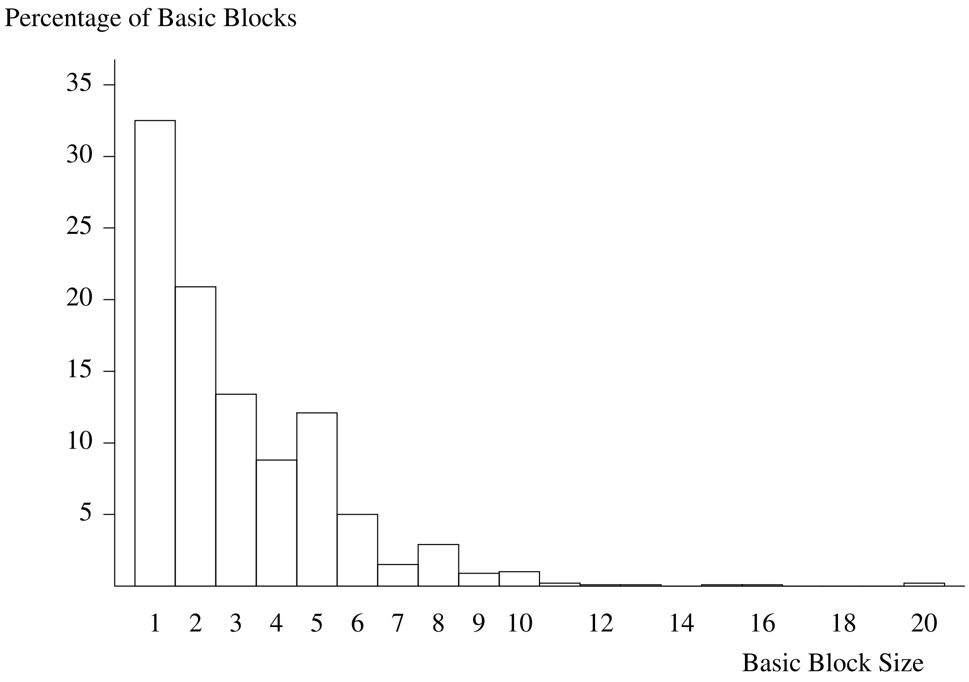


**Figure 6.1. Static Distribution of Basic Block Sizes**

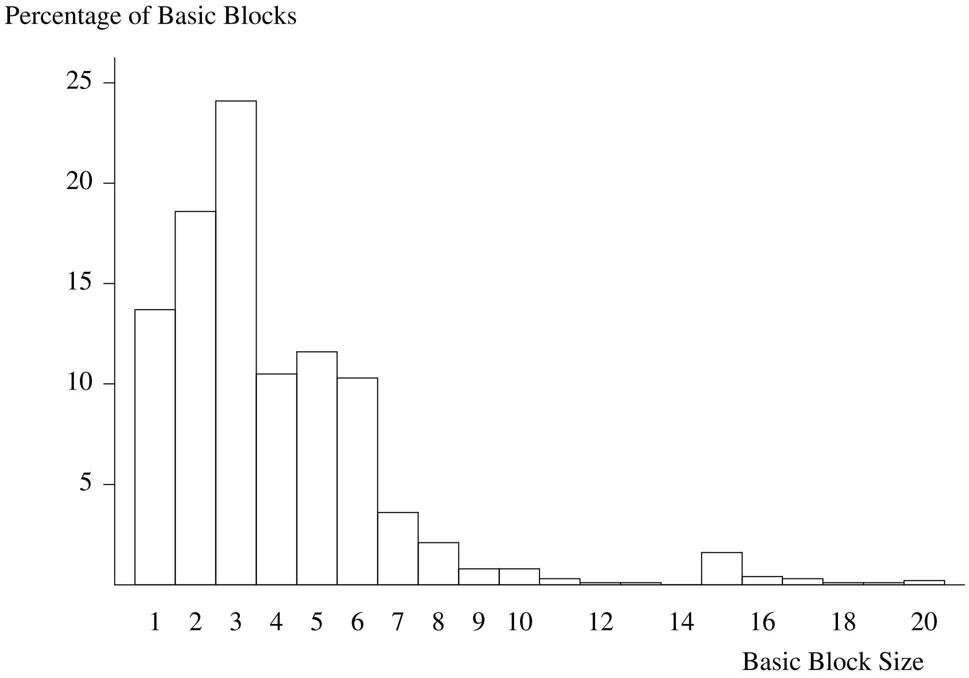


**Figure 6.2. Dynamic Distribution of Basic Block Sizes**

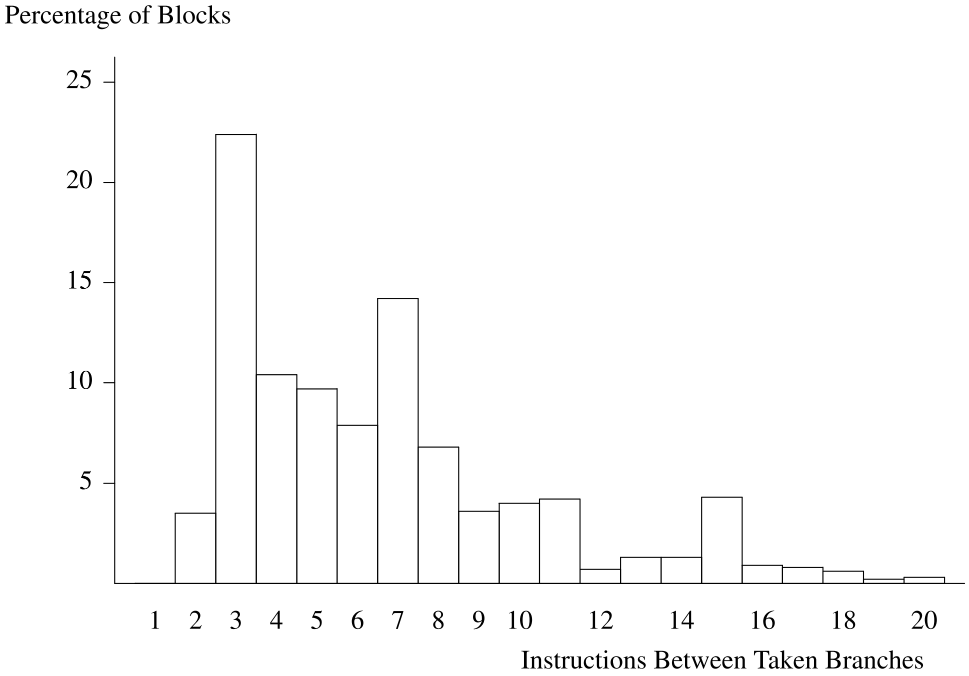


**Figure 6.3. Instructions Executed Between Taken Branches**

high frequencies at lower basic block sizes in these two figures suggests only a small potential for performance gains through instruction-level parallelism. Figure 7.3 shows the length of instruction sequences between taken branches, that is, the number of instructions that execute after one taken branch or jump instruction up until and including the next taken branch or jump instruction. The average length of sequences between taken branches is seven instructions.

### 6.1. Pipelined Implementation Characteristics

Execution times for the benchmarks for various pipeline depths were computed for two pipelined architectural models. Using the execution time of a non-pipelined implementation as a base for each model, speedups were calculated for each pipeline depth. The average of the speedups across the benchmarks are displayed in figure 6.4 for each pipeline depth and for each model.

The execution time for each benchmark is obtained by scheduling the instructions within its basic blocks taking pipeline stalls caused by data dependencies into account. This determines the number of pipeline stages needed

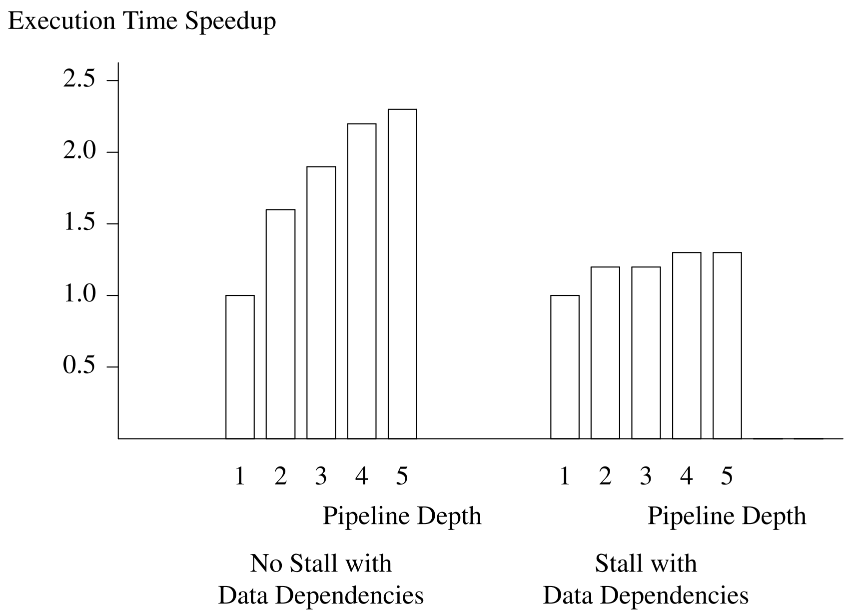


**Figure 6.4. Speedups Through Pipelined Implementations**

to execute each basic block. The number of stages needed to execute a benchmark is calculated by summing the products of the frequency of each basic block by the number of stages it needs to execute. By making the assumption that the cycle times of an implementation are independent of the number of pipeline stages in a cycle, (that is, by assuming that a single stage of a pipeline of depth N will take 1/N of the time of the cycle time of a non-pipelined machine), the relative execution time for each pipeline depth is computed.

The two architectural models differ in the length of data dependency pipeline stalls. Both models assume that a branch or jump instruction will stall the pipeline a single cycle (N stages). In the first model, data dependencies cause no stall at all. The first model gives the ideal maximum speedups taking into account only control dependencies. This model best approximates a pipelined machine with data forwarding and high hit-ratio. The second model assumes that the result of a computational operation takes a single cycle (N stages) before it is available for use in a subsequent instruction and that it takes two cycles (2N stages) before the value of a load instruction is available in a destination register. This is a realistic model for pipelined machines without data forwarding and with well-designed memory systems.

The speed-ups shown in figure 6.4 are less than ideal for both models. For the model which ignores data dependencies, pipelines of depth 2, 3, and 4 give speedups of only 1.6, 1.9, and 2.2. After three stages, the speedups for this model level off. This model shows that the frequencies of branch and jump instructions alone are significant limitations to pipelined speedups in Prolog. For the model which includes pipeline stalls for data dependencies, speedups are still lower and leveling off occurs almost immediately. Both pipelines of depths 2 and 3 give speedups of only 1.2.

## 6.2. Multiple Operation Issue Implementation Characteristics

Figure 6.5 and figure 6.6 display the average of the speedups across the benchmarks for four different multiple operation issue architectural models limiting the number of simultaneous operation issues at several levels and using a single operation issue, non-pipelined model as a base.

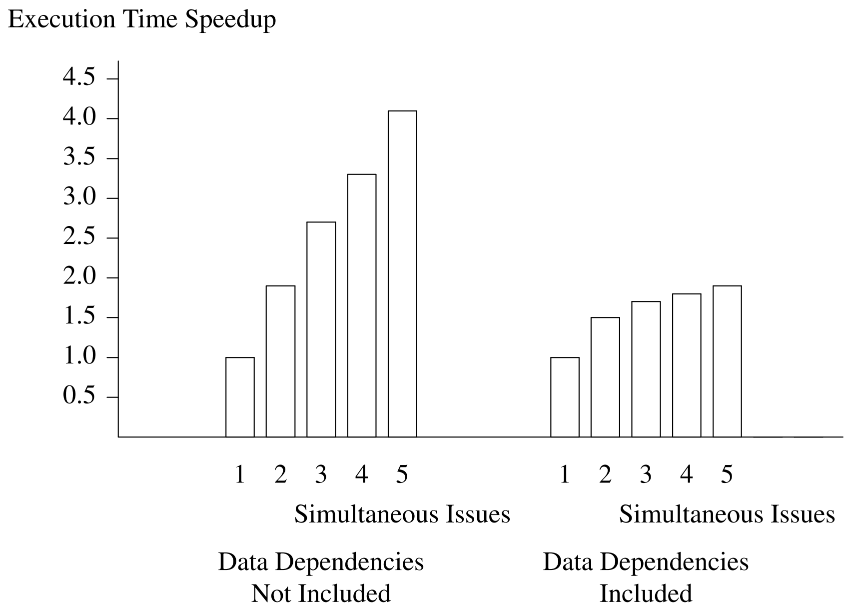


**Figure 6.5. Speedups Through Multiple Operation Issue (Arbitrary Operations)**

Execution times for each benchmark are obtained by scheduling the instructions within basic blocks taking data dependencies into account and disallowing out-of-order instruction execution. This determines the number of cycles needed to execute each basic block that is multiplied by the frequency of occurrence of that basic block. All such products are summed to obtain the number of cycles needed to execute a benchmark. By assuming equivalent cycle times for simultaneous operation issue levels, the relative execution times for each level of multiple operation issue is computed for each model.

Each of the figures display results for models similar to those used in the pipelined implementation discussion. In the first model in each figure, data dependencies are not considered when scheduling instructions. This is not a very practical model. In the second model in each figure, the result of a computational operation takes a single cycle and the result of a load operation takes two cycles to be placed in its destination register for use in a subsequent instruction. The difference between the two figures is in the type of operations that may be issued simultaneously. In figure 6.5, arbitrary operations may be issued during the same cycle. The results of figure 6.6 are obtained by

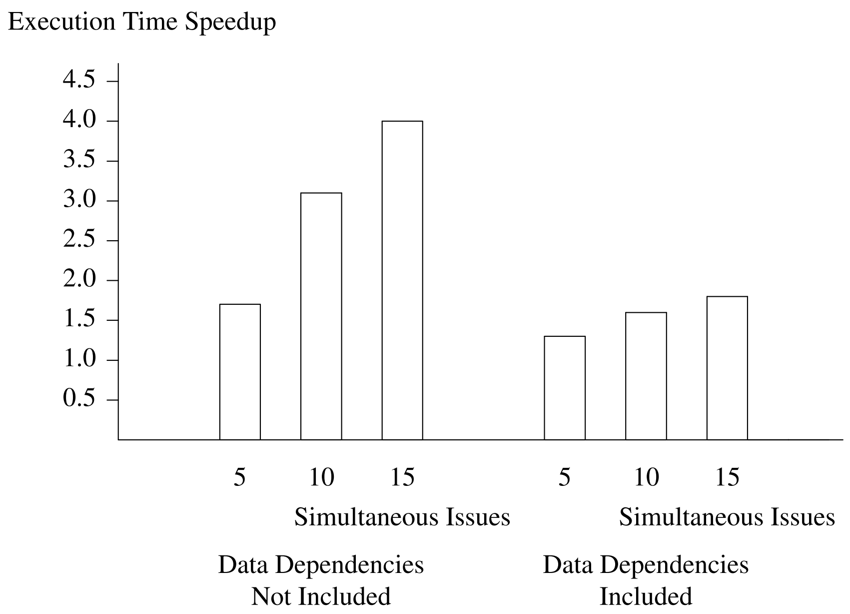


**Figure 6.6. Speedups Through Multiple Operation Issue (Partitioned Operations)**

partitioning operations into load, store, arithmetic, logical, and decision groups and limiting the number of operations that belong to the same group that can be simultaneously issued. For example, when five operations may be issued simultaneously in the models of the first figure, any five arbitrary operations may be simultaneously issued. When five operations may be issued simultaneously in the models of the second figure, each operation must belong to a different group. When ten operations may be issued concurrently in the models of the second figure, no more than two operations must belong to the same group. These two models reflect different machine organizations. The partitioned operation model most realistically reflects an implementation containing multiple hardware units and multiple datapaths divided by functionality. The arbitrary operation model reflects an implementation containing multiple hardware units with multiple datapaths in which each hardware unit is capable of executing the entire operation set of the machine.

Although less than ideal speedups, the multiple operation issue speedups obtained when ignoring data dependencies are almost linear; however, these models are not practical since there is no benefit to issuing two instructions at the same time if one instruction needs the result computed by a simultaneously issued instruction. The dependent instruction must wait for the completion of the instruction that computes its dependency before it may execute. simultaneously issued instruction. The arbitrary operation model that includes data dependencies has speedups of 1.5, 1.7, 1.8, and 1.9 for implementations designed for 2, 3, 4, and 5 simultaneously issued operations. The introduction of partitioning simultaneously issued operations causes a further and significant reduction in speedup. The partitioned operation model that includes data dependencies has a speedup of 1.3 for an implementation designed to allow the simultaneous issue of one operation from each of the five groups.

# CHAPTER 7

# Conclusions

## 7.1. Characterization and Optimization Results

We have presented many detailed characteristics of Prolog based on simulations with a new suite of benchmarks using the SIMPLE Prolog system which includes a compiler and simulator for a load-store instruction set architecture. These characteristics have proven useful in guiding the search for improvements to be made in the implementation of Prolog systems and provide a set of fundamental information that should be useful in exploring implementation issues of Prolog systems in the future.

We can make a few generalizations about the characteristics of Prolog execution by noting those attributes that are present in nearly all of our benchmarks. The process of unification, its degree of frequency and the operations that must be performed and data structures that must exist in order to realize this process is the most influential factor in a Prolog implementation. Nearly half of the execution of a Prolog program is spent within either a generic unification routine or within compiled unification code. The large amount of time spent carrying out the process of unification translates into processing a large number of tag-handling operations, tests for equality among objects, and data movements to create new objects. At a lower level, this translates into a large number of logical bitwise operations and many branch and jump instructions.

Built-in predicate operations in Prolog and their efficient implementation are more important to efficient Prolog execution than might be suspected. Built-in predicates provide more than a library of functions that may or may not be used in a specific program as with a procedural input/output or math library. Built-in predicates are much more fundamental in creating Prolog programs. This is reflected in the large amount of execution time spent within built-in predicate code.

We may also generalize by noting those attributes that differ most frequently between different benchmarks. Prolog programs differ in their use of different memory areas, in their selection of built-in predicates and the time spent in implementing search. The most important questions to consider when differentiating between the behavior of different Prolog programs are as follows. Is the program's reference pattern global stack intensive or local stack intensive or neither? Does the program require a particularly large space for either of these two areas? Is there a large use of built-in predicates or will there be a high use of general unification? If so, there will be a high demand on the push-down list. If not, compiled unification increases in importance. How much time will the benchmark spend managing search through its execution?

We have presented performance analyses obtained by simulating our benchmarks of improvements obtainable through a number of optimizations suited to Prolog implementation.

Several compiler optimization techniques have been found to contain large potential gains for both static code reduction and dynamic execution time improvements. Optimizations which make use of procedure argument mode information at compile time within compiled unification operations reduce our benchmarks' size by forty percent and decrease the number of instructions executed by nine percent. The most useful mode information is identification of read and write mode within unify operations for structure and list elements. Predetermining whether an argument is a variable or an instantiated object at compile time is beneficial as well. A smaller benefit is obtained by determining an argument's type or value at compile time. Knowledge of dereferencing characteristics reduces the benchmarks' static code size by six percent and reduces instructions executed within the simulations by six percent through eliminating dereferencing operations for objects which are directly referenced and by eliminating dereference operations when an object has previously been dereferenced and this dereferenced value may be made substituted for the non-dereferenced value. Optimization of arithmetic expression representation and arithmetic expression evaluation eliminates six percent of the benchmarks' executed instructions which create arithmetic expressions and eliminates seven percent of the benchmarks' executed instructions which evaluate these arithmetic structures. Arithmetic expression optimization decreases the static code size of the benchmarks by ten percent.

Performance improvements may be obtained with minor modifications to a general-purpose load-store instruction set architecture. Architectural support for tag-handling operations through design and use of an architecture which recognizes the existence of tag and value fields within an object leads to a twelve percent decrease in static code size and a seven percent decrease in executed instructions in the benchmark simulations. The addition of orthogonal tag instructions reduces code size by an additional nine percent and reduces dynamic instructions by eleven percent. Allowing instructions which operate solely on tag fields to execute in parallel with instructions which operate solely on value fields results in a two percent overlapping of instructions. The introduction of push and pop instructions decreases the benchmarks' code size by seven percent and the number of instructions executed by eight percent.

The most important features for which to add support in a general-pupose machine in order to obtain significant improvements in Prolog execution appear to be support for dynamic data types and dynamic typing and support for non-determinism or backtracking. There appear to be harmonious techniques which assist these features at some level with minimal disruption of a general-purpose machine. Language-specific compiler optimizations are very beneficial when implementing Prolog. Architectural support for dynamic data types and dynamic typing in Prolog is similar to support for dynamic objects in Lisp. Significant support of backtracking appears to be more difficult to assist in a general-purpose fashion.

### 7.2. Increasing Instruction-Level Parallelism

We have found Prolog to attain only modest speedups through simple pipelined and multiple-operation-issue machine techniques.

Many enhancements have been developed for improving the performance of pipelined and multiple operation issue general-purpose architectures by reducing the effects of data and control dependencies. These include compiler analysis techniques such as instruction reordering, static branch prediction, loop unrolling, and trace scheduling, as well as architectural and implementation enhancements such as dynamic instruction scheduling, data forwarding, delayed loads and delayed branches, dynamic branch prediction, additional logic for fast branch decision computation, and multithreading [58, 109]. Some of these enhancements are very appropriate for Prolog. (We have calculated that compile-time branch prediction can successfully predict as many as eighty-five percent of branch outcomes.) Others seem less appropriate for Prolog. (Loop unrolling is an enhancement that would be of little benefit.) Enhancements specifically developed for Prolog that increase instruction-level parallelism are also possible.

We intend to continue our research on efficient Prolog implementation by investigating techniques that will increase the instruction-level parallelism achieved in the execution of Prolog programs by identifying and developing techniques specifically aimed at the Prolog language. In particular, we believe there are many higher-level and lower-level characteristics of Prolog that can be exploited by building an appropriate abstract execution model that aims to identify groups of potentially concurrent operations expanded beyond the basic block. A Prolog compiler should take advantage of its knowledge of the semantics of the Prolog language to increase actual instruction-level parallelism.

Characteristics of Prolog that may lead to an increase of instruction-level parallelism include the following. The separate components of tag and value fields within an object allows operations whose operands are restricted to either tag or value fields to execute concurrently. Although the overlapping of these types of instructions computed previously seemed of limited benefit, the previous calculations were performed for an implementation that was already specially enhanced with support for operations on tagged objects. The tagged-operand operations resulted in a removal of many tagged-operand instructions. An architecture that does not include this level of tag support would have a significantly greater amount of tag and value operation overlap.

Although unification has been theoretically shown to be a fundamentally sequential operation, it is possible to shift and overlap suboperations from the unifications that occur for different arguments when beginning the execution of a clause. It is possible to overlap the creation of a choice point when a procedure is first started with the unification operations of the first selected clause. Unification operations occurring when alternate clauses are executed can begin prematurely as the recovery of state information (that is, backtracking) completes from the failure that caused the alternate clause to be initiated.

The area of abstract interpretation which has already proven its worth in assisting global optimization in Prolog compilation and the emerging area of multi-threaded architectures that aims to increase the achieved level of instruction-level parallelism are two areas with concepts that may combine nicely to refine the above ideas and develop new techniques for attaining significant instruction-level parallelism during Prolog execution. We intend to use use abstract interpretation to identify operations that may execute with some degree of concurrency and schedule concurrent operations within a multi-threaded architecture. It is our hope that the combination of these two techniques will take advantage of the parallelism that seems to be available in Prolog and in logic programming languages in general that has not been satisfactory exploited as yet.. While research in parallel logic programming languages has had difficulties in located large-scale parallelism, the same degree of parallelism is not at all necessary for success at the instruction level. As general-purpose processors continue to increase the use of instruction-level parallelism and general-purpose languages continue to benefit from these enhancements, future research into continuing to advance the implementations of languages such as Prolog that do not fit as neatly into "general-purpose" schemes becomes increasingly important if we wish to make use of the programming superiority of these languages without losing the seemingly performance advantages of traditional languages.

### 7.3. Conclusions

It is our belief that Prolog is an important foundation for languages of the future. For many traditional, recently developed, and most important, emerging applications, Prolog is a language superior to others because of its high-level expressiveness. Prolog's declarative style of programming, search and pattern-matching capabilities, use of dynamic data types, conciseness, conversational interaction, and potential for parallelism are important features which will appear in future languages as they are designed. Today, Prolog has a dedicated base of users. Commercial activity is well underway to enrich its programming environment with the capabilities expected of an industrial-strength programming language. Industry has made a commitment to developing new Prolog applications. Tomorrow, Prolog's use can be expected to increase as the next decade sees an increasing number of symbolic applications.

Prolog has unique characteristics which challenge an efficient implementation which result from its use of dynamic data structures, dynamic types, searching, and unification. It is without doubt throughout the foreseeable future that procedural languages will continue to drive computer architecture research. It is then the obligation of researchers of Prolog implementations to continue to make improvements to Prolog systems by exploiting enhancements to general-purpose architectures, (which are labeled as general purpose through the dominance of procedural languages and numerical applications), through modest changes that can be incorporated into a general-purpose architecture without impacting the performance of its more common uses, and through the use of optimizing compilers which complement the architecture, expose and refine lower-level details of the execution model, and utilize knowledge of the semantics of the language. We have illustrated that by analyzing architectural-level details to identify striking or noticeably different characteristics of a benchmarked simulation, one can propose and quantify improvements for modest enhancements to an architecture and adjustments to the abstract machine model to add significant support for Prolog on general-purpose, load-store instruction set machines.

As general-purpose machines continue to be enhanced, Prolog implementation research continues to be challenged to keep pace with performance improvements seen in more traditional languages. Because we believe it is essential for languages such as Prolog which encourage new ways to think and express computational ideas, which suggest new directions for increasing programming productivity, and which introduce new potentials for performance improvement, we believe it is essential that research such as ours continue into the future.

# References


[1] K. Furukawa, R. Nakajima, A. Yonezawa, S. Goto, and A. Aoyama, "Problem Solving and Inference Mechanisms," in *Proceedings of the International Conference on Fifth Generation Computer Systems*. Tokyo, Japan, pp. 131-138, October 1981.

[2] K. Furukawa and T. Yokoi, "Basic System Software," in *Proceedings of the International Conference on Fifth Generation Computer Systems*. pp. 37-57, 1984.

[3] K. Fuchi and K. Furukawa, "The Role of Logic Programming in the Fifth Generation Computer Systems Project," in *Proceedings of the Third International Conference on Logic Programming*. Berlin, Germany, pp. 1-24, 1986.

[4] K. Furukawa, "Fifth Generation Computer Project: Current Research Activity and Future Plans," in *Proceedings of the International Joint Conference on Theory and Practice of Software Development*. Pisa, Italy, March 1987.

[5] T. P. Dobry, A. M. Despain, and Y. N. Patt, "Performance Studies of a Prolog Machine Architecture," in *Proceedings of the 12th Annual International Symposium on Computer Architecture*. pp. 180-190, December 1985.

[6] J. W. Mills, "A High Performance LOW RISC Machine for Logic Programming," *Journal of Logic Programming*, vol. 6, pp. 179-212, 1989.

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

[8] R. A. Kowalski, "Algorithm = Logic + Control," *Communications of the ACM*, vol. 22, pp. 424-436, July 1979.

[9] R. A. Kowalski, in *Logic for Problem Solving*. Amsterdam, Holland: North Holland, 1979.

[10] J. A. Robinson, "Logic Programming -- Past, Present and future," *Journal of New Generation Computing*, vol. 1, pp. 107-124, 1983.

[11] R. A. Kowalski, "The Early Years of Logic Programming," *Communications of the ACM*, vol. 31, pp. 38-43, January 1988.

[12] J. Cohen, "A View of the Origins and Development of Prolog," *Communications of the ACM*, vol. 31, pp. 26-37, January 1988.

[13] L. Sterling, W. F. Clocksin, and C. S. Mellish, in *Programming in Prolog*. Berlin, Germany: Springer-Verlag, 1981.

[14] L. Sterling and E. Shapiro, in *The Art of Prolog*. Cambridge, Massachusetts: MIT Press, 1986.

[15] A. Colmerauer, "Prolog in 10 Figures," *Communications of the ACM*, vol. 28, pp. 1296-1324, December 1985.

[16] D. H. Warren, "Logic Programming and Compiler Writing," *Software - Practice and Experience*, vol. 10, pp. 97-125, 1980.

[17] F. Pereira, "Logic for Natural Language Analysis," Technical Note 275, Artificial Intelligence Center, SRI International, Menlo Park, California, January 1983.

[18] A. Porto and M. Filgueiras, "Natural Language Semantics: A Logic Programming Approach," in *Proceedings of the 1984 International Symposium on Logic Programming*. Atlantic City, New Jersey, pp. 228-232, February 1984.

[19] V. Dahl and P. Saint-Dizier, in *Natural Language Understanding and Logic Programming*. Amsterdam, Holland: North Holland, 1985.

[20] F. Pereira and S. M. Shieber, in *Prolog and Natural-Language Analysis*. Stanford, California: Center for the Study of Language and Information, 1987.

[21] A. Littleford, "A MYCIN-like Expert System in Prolog," in *Proceedings of the Second International Logic Programming Conference*. Uppsala, Sweden, pp. 289-300, 1984.

[22] D. R. Brough and I. F. Alexander, "The Fossil Expert System," *Expert Systems*, vol. 3, pp. 76-83 .

[23] D. B. Searls and K. M. Norton, "Logic-Based Configuration with a Semantic Network," *Journal of Logic Programming*, vol. 8, pp. 53-73, 1990.

[24] C. J. Rawlings, W. R. Taylor, J. Nyakairu, J. Fox, and M. J. E. Sternberg, "Using Prolog to Represent and Reason about Protein Structure," in *Proceedings of the Third International Conference on Logic Programming*. London, United Kingdom, pp. 536-543, July 1986.

[25] C. Fellenstein, C. Green, L. Palmer, A. Walker, and D. Wyler, "A Prototype Manufacturing Knowledge Base in SYLLOG," *IBM Journal of Research and Development*, vol. 29, pp. 413-421, 1985.

[26] H. Gallaire, J. Minker, and J. Nicolas, "Logic and Databases: A Deductive Approach," *ACM Computing Surveys*, vol. 16, pp. 153-185, June 1984.

[27] J. W. Lloyd and R. W. Topor, "A Basis for Deductive Database Systems," *Journal of Logic Programming*, vol. 2, pp. 93-109, 1985.

[28] C. Zaniolo, "Prolog: A Database Query Language for All Seasons," in *Proceedings of the First International Workshop on Expert Database Systems*. Kiawah Island, South Carolina, October 1984.

[29] S. Ceri, G. Gottlob, and G. Wiederhold, "Interfacing Relational Databases and Prolog Efficiently," in *Proceedings of the First International Conference on Expert Database Systems*. Charleston, South Carolina, pp. 141-153, April 1986.

[30] Y. Ioannidis, J. Chen, M. A. Friedman, and M. Tsangaris, "BERMUDA - An Architectural Perspective on Interfacing Prolog to a Database Machine," in *Proceedings of the Expert Database Systems Conference*. 1988.

[31] R. Gupta, "Test-Pattern Generation for VLSI Circuits in a Prolog Environment," in *Proceedings of the Third International Conference in Logic Programming*. pp. 528-535, July 1986.

[32] W. R. Bush, G. Cheng, P. C. McGeer, and A. Despain, "Experience with Prolog as a Hardware Specification Language," in *Proceedings of the 1987 Symposium on Logic Programming*. San Francisco, California, August 1987.

[33] W. F. Clocksin, "Logic Programming and Digital Circuit Analysis," *Journal of Logic Programming*, vol. 4, pp. 59-82, 1987.

[34] P. B. Reintjes, "Aunt: A Universal Netlist Translator," *Journal of Logic Programming*, vol. 8, pp. 5-19, 1990.

[35] J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," *Journal of the ACM*, vol. 12, pp. 23-44, 1965.

[36] N. J. Nilsson, in *Principles of Artificial Intelligence*. Berlin, Germany: Springer-Verlag, 1981.

[37] J. W. Lloyd, in *Foundations of Logic Programming*. Berlin, Germany: Springer-Verlag, 1984.

[38] D. H. Warren, "Applied Logic -- Its Use and Implementation as a Programming Tool," University of Edinburgh, 1977, PhD Dissertation.

[39] D. H. Warren, "An Abstract Prolog Instruction Set," Technical Note 309, Artificial Intelligence Center, SRI International, Menlo Park, California, October 1983.

[40] E. Tick and D. H. Warren, "Towards a Pipelined Prolog Processor," in *Proceedings of the 1984 International Symposium on Logic Programming*. pp. 29-40, 1984.

[41] T. P. Dobry, Y. N. Patt, and A. M. Despain, "Design Decisions Influencing the Microarchitecture for a Prolog Machine," in *Proceedings of the 17th Annual Microprogramming Workshop of the IEEE Computer Society*. October 1984.

[42] 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.

[43] G. Borriello, A. R. Cherenson, P. B. Danzig, and M. N. Nelson, "RISCS vs. CISCS for Prolog: A Case Study," in *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*. Palo Alto, California, October 1987.

[44] D. A. Patterson and C. H. Sequin, "A VLSI RISC Computer," *IEEE Computer Magazine*, vol. 15, pp. 8-21, September 1982.

[45] G. Radin, "The 801 Minicomputer," *IBM Journal of Research and Development*, vol. 27, pp. 237-246, May 1983.

[46] J. L. Hennessy, N. Jouppi, F. Baskett, and J. Gill, "The MIPS Machine," in *Digest of Papers Compcon Spring 1982*. pp. 2-7, February 1982.

[47] "HP Precision Architecture and Instruction Set," Manual Part Number: 09740-90014, Hewlett-Packard Company, Palo Alto, California, April 1989.

[48] "Series 10000 Technical Reference Library: Processors and Instruction Set," Order Number: 011720-A00, Apollo Computer Inc., Chelmsford, Massachussetts, September 1988.

[49] C. S. Mellish, "Some Global Optimizations for a Prolog Compiler," *Journal of Logic Programming*, vol. 1, pp. 43-66, 1985.

[50] C. S. Mellish, "Abstract Interpretation of Prolog Programs," in *Proceedings of the Third International Conference on Logic Programming*. London, United Kingdom, pp. 463-474, July 1986.

[51] S. K. Debray and D. S. Warren, "Automatic Mode Inference for Logic Programs," *Journal of Logic Programming*, vol. 5, pp. 207-229, 1988.

[52] S. K. Debray and D. S. Warren, "Detection and Optimization of Functional Computations in Prolog," in *Proceedings of the Third International Conference on Logic Programming*. London, United Kingdom, pp. 490-504, July 1986.

[53] P. Van Roy, "Improving the Execution Speed of Compiled Prolog with Modes, Clause Selection, and Determinism," in *Proceedings of the International Joint Conference on Theory and Practice of Software Development*. Pisa, Italy, pp. 111-125, March 1987.

[54] P. Van Roy, "An Intermediate Language to Support Prolog's Unification," in *Proceedings of the North American Conference on Logic Programming*. MIT Press, pp. 1148-1164, October 1989.

[55] P. Van Roy and A. M. Despain, "The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler," in *Proceedings of the North American Conference on Logic Programming*. 1990.

[56] H. Touati and A. Despain, "An Empirical Study of the Warren Abstract Machine," in *Proceedings of the 1987 Symposium on Logic Programming*. San Francisco, California, pp. 192-204, August 1987.

[57] R. Ona, H. Shimuzu, K. Masuda, and M. Aso, "Analysis of Sequential Prolog Programs," *Journal of Logic Programming*, vol. 2, pp. 119-141, July 1986.

[58] J. L. Hennessy and D. A. Patterson, in *Computer Architecture: A Quantitative Approach*. San Mateo, California: Morgan Kaufmann Publishers, Inc., 1989.

[59] P. Steenkiste and J. Hennessy, "Lisp on a Reduced-Instruction-Set Processor," *Proceedings of the 1986 Conference on Lisp and Functional Programming*, pp. 192-201, August 1986.

[60] P. Steenkiste and J. Hennessy, "Lisp on a Reduced-Instruction-Set Processor: Characterization and Optimization," *IEEE Computer Magazine*, vol. 21, pp. 34-44, August 1988.

[61] E. Tick, "Memory Performance of Lisp and Prolog Programs," in *Proceedings of the Third International Conference on Logic Programming*. London, United Kingdom, pp. 642-649, July 1986.

[62] P. Van Roy, "A Prolog Compiler for the PLM," Report No. UCB/CSD 84/203, University of California, Berkeley, California, November 1984, Master's Thesis.

[63] D. A. Patterson, "Reduced Instruction Set Computers," *Communications of the ACM*, vol. 28, pp. 8-21, January 1985.

[64] J. Hennessy, "VLSI RISC Processors," *VLSI Systems Design*, vol. 1, pp. 22-32, October 1985.

[65] E. Tick, "An Overlapped Prolog Processor," Technical Note 308, Artificial Intelligence Center, SRI International, Menlo Park, California, October 1983.

[66] K. Taki, M. Yokota, A. Yamamoto, H. Nishikawa, S. Uchida, H. Nakashima, and A. Mitsuishi, "Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI)," in *Proceedings of the International Conference on Fifth Generation Computer Systems*. 1984.

[67] D. H. D. Warren and L. M. Perreira, "Prolog - The Language and its Implementation Compared with Lisp," in *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*. pp. 109-115, August 1977.

[68] D. S. Warren, "The Runtime Environment for a Prolog Compiler using a Copy Algorithm," Technical Report 83/052, Computer Science Department, SUNY at Stony Brook, Stony Brook, New York, March 1984.

[69] H. Ait-Kaci, "The WAM: A (Real) Tutorial," PRL Research Report 5, Paris Research Laboratory, Digital Equiptment Corporation, Paris, France, 1990.

[70] H. Nishikawa, M. Yokota, A. Yamamoto, K. Taki, and S. Uchida, "The Personal Sequential Inference Machine (PSI) : Its Design Philosophy and Machine Architecture," *Proceedings of the Logic Programming Workshop*, pp. 53-72, 1983.

[71] S. Abe, T. Bandoh, S. Yamaguchi, K. Kurosawa, and K. Kiriyama, "High Performance Integrated Prolog Processor IPP," in *Proceedings of the 14th International Symposium on Computer Architecture*. Pittsburgh, Pennsylvania, pp. 100-107, June 1987.

[72] M. Morioka, S. Yamaguchi, and T. Bandoh, "Evaluation of Memory System for Integrated Prolog Processor IPP," in *Proceedings of the 16th International Symposium on Computer Architecture*. Jerusalem, Israel, pp. 203-221, May 1989.

[73] H. Benker, J. M. Beacco, S. Bescos, M. Dorochevsky, T. Jeffre, A. Pohimann, J. Noye, B. Poterie, A. Sexton, J. C. Syre, O. Thibault, and G. Watzlawik, "KCM: A Knowledge Crunching Machine," in *Proceedings of the 16th International Symposium on Computer Architecture*. Jerusalem, Israel, pp. 186-194, May 1989.

[74] A. M. Despain and Y. N. Patt, "Aquarius: A High Performance Computing System for Symbolic/Numeric Applications," in *Digest of Papers Compcon Spring 1985*. IEEE Press, pp. 376-382, February 1985.

[75] A. M. Despain, Y. N. Patt, T. P. Dobry, J. H. Chang, and W. Citrin, "High Performance Prolog, The Multiplicative Effect of Several Levels of Implementation."

[76] V. P. Srini, J. V. Tam, T. M. Nguyen, Y. N. Patt, A. M. Despain, M. Moll, and D. Ellsworth, "A CMOS Chip for Prolog," in *Proceedings fo the International Conference on Computer Design*. pp. 605-610, October 1987.

[77] K. Kawanobe, "Current Status and Future Plans of the Fifth Generation Computer Systems Project," in *Proceedings of the International Conference on Fifth Generation Computer Systems*. pp. 3-17, 1984.

[78] K. Furukawa, S. Kunifuji, A. Takeuchi, and K. Ueda, "The Conceptual Specification of the Kernel Language Version 1," ICOT TR-054, Institute for New Generation Computer Technology, 1984.

[79] M. Yokota, T. Hattori, and T. Chikayama, in *Fifth Generation Kernel Language*. Institute for New Generation Computer Technology, October 1982.

[80] M. Yokota, A. Yamamoto, K. Taki, H. Nishikawa, and S. Uchida, "The Design and Implementation of a Personal Sequential Inference Machine: PSI," *Journal of New Generation Computing*, vol. 1, pp. 125-144, 1983.

[81] K. Nakajima, H. Nakashima, M. Yokota, K. Taki, S. Uchida, H. Nishikawa, A. Yamamoto, and M. Mitsui, "Evaluation of PSI Micro-Interpreter," in *Digest of Papers Compcon Spring 1986*. San Francisco, California, May 1986.

[82] K. Taki, K. Nakajima, H. Nakashima, and M. Ikeda, "Performance and Architectural Evaluation of the PSI Machine," in *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*. Palo Alto, California, October 1987.

[83] C. S. Mellish, "An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter."

[84] M. Hill, et al., "Design Decisions in SPUR," *IEEE Computer Magazine*, vol. 19, pp. 8-22, November 1986.

[85] J. W. Mills, "Coming to Grips with a RISC: A Report of the Progress of the LOW RISC Design Group," *ACM SIGARCH Computer Architecture News*, vol. 15, pp. 53-62, March 1987.

[86] B. Short, "Use of Instruction Set Simulators to Evaluate the LOW RISC," *ACM SIGARCH Computer Architecture News*, vol. 15, pp. 63-67, March 1987.

[87] D. Ungar, "Architecture of SOAR: Smalltalk on a RISC," in *Proceedings of the Eleventh International Symposium on Computer Architecture*. June 1988.

[88] P. Kursawe, "How to Invent a Prolog Machine," in *Proceedings of the Third International Conference on Logic Programming*. London, United Kingdom, pp. 134-148, July 1986.

[89] M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen, "Abstract Interpretation Towards the Global Optimization of Prolog Programs," in *Proceedings of the 1987 Symposium on Logic Programming*. San Francisco, California, pp. 192-204, August 1987.

[90] A. K. Turk, "Compiler Optimizations for the WAM," in *Proceedings of the Third International Conference on Logic Programming*. London, United Kingdom, pp. 657-662, July 1986.

[91] T. Hickey and S. Mudambi, "Global Compilation of Prolog," *Journal of Logic Programming*, vol. 7, pp. 193-230, 1989.

[92] U. S. Reddy, "Transformation of Logic Programs into Functional Program," in *Proceedings of the 1984 International Symposium on Logic Programming*. Atlantic City, New Jersey, pp. 187-196, February 1984.

[93] W. Drabent, "Do Logic Programs Resemble Programs in Conventional Languages?," in *Proceedings of the 1987 Symposium on Logic Programming*. San Francisco, California, August 1987.

[94] H. Mannila and E. Ukkonen, "Flow Analysis of Prolog Programs," in *Proceedings of the 1987 Symposium on Logic Programming*. San Francisco, California, August 1987.

[95] S. K. Debray, "Flow Analysis of Dynamic Logic Programs," *Journal of Logic Programming*, vol. 7, pp. 149-176, 1989.

[96] G. Kildall, "A Unified Approach to Global Program Optimization," in *Proceedings of the First Symposium on Principles of Programming Languages*. pp. 194-206, January 1973.

[97] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Proceedings of the Fourth Symposium on Principles of Programming Languages*. pp. 238-252, January 1977.

[98] M. A. Covington, D. Nute, and A. Vellino, in *Prolog Programming in Depth*. Glenview, Illinois: Scott, Foresman and Company, 1988.

[99] I. Bratko, in *Prolog Programming for Artificial Intelligence*. Reading, Massachusetts: Addison-Wesley, 1986.

[100] P. M. Ross, in *Advanced Prolog: Techniques and Examples*. Reading, Massachusetts: Addison-Wesley, 1989.

[101] A. Newell and H. A. Simon, in *Human Problem Solving.*. Englewood Cliffs, New Jersey: Prentice-Hall, 1972.

[102] D. Knuth, in *The Art of Computer Programming*. p. 394, 1981, 2nd Edition.

[103] E. Tick, "Prolog Memory-Referencing Behavior," Technical Report No. 85-281, Standford University, Stanford, California, September 1985.

[104] E. Tick, "Studies in Prolog Architectures," Technical Report No. CSL-TR-87-329, Standford University, Stanford, California, June 1987, PhD Disertation.

[105] P. M. Kogge, in *The Architecture of Pipelined Computers*. New York: McGraw-Hill, 1981.

[106] S. R. Kunkel and J. E. Smith, "Optimal Pipelining in Supercomputers," in *Proceedings of the 13th Annual International Symposium on Computer Architecture*. pp. 404-411, June 1986.

[107] T. Agerwala and J. Cocke, "High Performance Reduced Instruction Set Processors," IBM Technical Report, March 1987.

[108] J. A. Fischer, "Very Long Instruction Word Architectures and ELI-512," in *Proceedings of the 10th Symposium on Computer Architecture*. pp. 478-490, June 1983.

[109] G. M. Papadopoulos and K. R. Traub, "Multithreading: A Revisionist View of Dataflow Architectures," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*. pp. 342-351, May 1991.

# APPENDIX A

## Representation of Fact-Intensive Programs

In the WAM model, facts are treated as rules and are compiled into SIMPLE instructions. In chapter 3, we explored the percentage that facts contribute to static code size and found grammar, statistics, routes, compiler and hexconversion were fact-intensive benchmarks. Facts in the grammar and statistics benchmarks weigh equally with rules in contributing to static code size with respect to their percentage in the distribution of clauses. In other fact-intensive benchmarks, the percentage of facts with respect to total clauses is somewhat larger than the percentage of facts with respect to static code size; however, the static code percentage is still of significant size. Fact procedures are predominantly sequences of get and unify operations. Each get and unify operation performs dereferencing, perhaps decdring, determines and reacts to the mode of a term passed to its procedure, and then creates an object to which the term is instantiated or verifies that the term matches some object. Due to the complexity of get and unify operations, programs that contain a large number of facts, (for example, large database applications), compile to large SIMPLE programs.

Consider an example database application containing one thousand records naming individuals and their parents. This information might be represented as a table of one thousand entries of two one-word pointers to names in a symbol table using a traditional language. In Prolog, the same information would be represented by facts of the form `parent(parent_name,child_name).` A fact of this form translates to four WAM operations and requires seventy-one SIMPLE instructions as shown in figure A.1. The entire database would be represented by over seventy thousand SIMPLE instructions (plus a symbol table) or over thirty-five times the size of the two-thousand-word table in the procedural language.

Representing facts in the WAM model shows one extreme in the tradeoff of Prolog implementation. The primary benefits of the WAM are achieved by implementing unification as compiled code to obtain a faster program, but at the expense of a significantly larger representation of the program's content. The opposite extreme is interpre-

Prolog Source Code

```
parent(parent_name, child_name).
```

| WAM Operations | Static SIMPLE Instructions |
|---|---|
| `retry_me_else <next_clause>` | 3 |
| `get_atom X1,parent_name` | 32 |
| `get_atom X2,child_name` | 32 |
| `proceed` | 2 |
| | 71 |

**Figure A.1. SIMPLE Static Code Size for Fact Representation**

tation that compactly represents data but must use a slower generic unification technique. Representation of facts is often an extreme case of the WAM's liability of including the full flexibility and power of Prolog's semantics in the code it produces despite contexts in which the full features of the language are not utilized. For some fact-intensive applications, the pure WAM representation of facts is clearly unacceptable.

Prolog implementation researchers and certainly WAM implementation researchers have not identified the specific problem of the unreasonable expansion of static code size when representing a large database of facts although efforts in other areas indirectly benefit this problem. Deductive database systems, motivated from a broader perspective predominantly by the database community, have been proposed to bridge the gap between logic programming and database technology. These systems address the need to enhance database systems with inferencing capabilities and exploit benefits gained by coupling database systems with logic programming systems. Although logic programming and database technology are both founded on first-order predicate logic, efforts to combine these two types of systems has suffered from inconsistencies in their semantics. Existing designs have not achieved adequate success from a performance point of view. Often these systems result in a duplication of storage and a duplication of search effort within each of the components of the combined system. Regardless, the aim of a deductive database system is not as much to develop a general-purpose logic programming language as it is to develop an intelligent database system.

Many global analysis and optimization techniques generally applicable for WAM optimization are beneficial to reducing the static code size of fact representation. Optimizations through mode analysis eliminate unnecessary operations throughout compiled unification code. Decision graph structuring, as developed in the BAM model of execution, can move common operations contained within each of the facts of a predicate to a single shared section of code. For example, the code within each fact clause of a predicate may begin with the dereferencing of the first argument. It is possible that these replicated code fragments can be combined and placed before the code that decides on the selection of clauses.

A third alternative to reduce the large static code of a fact-intensive Prolog program is to sacrifice some performance benefits achieved through the WAM model and increase the use of general unification through a combined WAM-database model that is appropriate for programs containing many ground clauses. In the combined WAM-database model, WAM representations for ground terms within facts are created at compile time and placed on the top of the global stack. During the execution of a fact, each procedure argument is unified by pushing its address on the push-down list, pushing the address of the compiler-created term corresponding to this argument on the push-down list, and calling a general unification routine. Some optimization within the scheme are possible. In the case of a constant term within the fact definition, the constant itself can be pushed on the push-down list by in-line instructions. An object need not be created on the global stack at compile time. The "general" unification routines(s) called for this use may be optimized in that it is known that the first object to be unified is either a dereferenced constant or an address that immediately references a list or structure of ground terms. Figure A.2 illustrates how combining the WAM and database representations can reduce a fact of the form `parent(parent_name, child_name)` that was shown to contain seventy-one SIMPLE instructions in a pure WAM model to eight SIMPLE instructions in the new model. The number of instructions executed for each fact increases with this scheme because of the overhead of the procedure call to a general unification routine and the substitution of special-case unification with a less efficient and more generic unification.

Table A.1 shows the decrease in static code size and the increase in instructions executed when this technique is applied to the five fact-intensive benchmarks. Four of these benchmarks realize significant reductions in code size ranging from fourteen percent to forty-five percent with a modest increase in dynamic instructions ranging from one percent to seven percent. The code size reductions with respect to the static code related to fact representation for

Prolog Source Code

```
parent(parent_name, child_name).
```

| Alternative Fact Representation | Static SIMPLE Instructions |
|---|---|
| `retry_me_else <next_clause>` | 3 |
| `push parent_name on push-down list` | 2 |
| `push child_name on push-down list` | 2 |
| `escape fact_unification` | 1 |
| | 8 |

**Figure A.2. Combined WAM-Database Fact Representation**

| Benchmark | Percentage of SIMPLE Instructions | |
|---|---|---|
| | Code Size Reduction | Dynamic Instruction Increase |
| compiler | 16.1 | 6.2 |
| grammar | 1.6 | 0.3 |
| hexconversion | 13.5 | 1.4 |
| routes | 25.7 | 6.6 |
| statistics | 44.7 | 3.1 |
| arithmetic mean | 20.3 | 3.5 |

**Table A.1. Performance Changes with WAM-Database Fact Representation**

these four benchmarks averages from seventy-six percent. The increase in executed instructions with respect to executed instructions related to fact representation for these four benchmarks averages twenty-five percent.

Only ground facts or variable-free fact definitions are suitable for this enhancement. The data objects created at compile time are read-only objects since they must be reusable by different fact invocations. Facts that contain variables must dynamically create these variables that are local to each invocation. Although the grammar benchmark has the largest percentage of fact-related code, it is applicable for use with the combined WAM-database implementation the least among the fact-intensive benchmarks because few of its facts are ground clauses.

# Abstract

We analyze the execution of a new suite of medium-sized and realistic benchmarks through simulations on a general-purpose, register-oriented architecture at the abstract machine and architectural levels to identify the most critical characteristics of Prolog for efficient execution. We propose improvements through modest enhancements to the architecture and to the Warren abstract machine to add significant support for the identified issues. Architectural support for tag-handling operations through an architecture which distinguishes between the tag and value fields of an object leads to a twelve percent decrease in static code and a seven percent decrease in executed instructions. The addition of orthogonal tag instructions reduces code size by an additional nine percent and reduces dynamic instructions by eleven percent. The introduction of push and pop instructions decreases code size by seven percent and the number of instructions executed by eight percent. Static code is reduced by forty percent and the number of instructions executed decreases by nine percent by utilizing procedure argument mode information at compile time within compiled unification operations. Knowledge of dereferencing characteristics reduces static code by six percent and reduces instructions executed by six percent by eliminating dereferencing for objects which are directly referenced and when previously dereferenced values may be substituted for non-dereferenced objects. Optimization of arithmetic expression evaluation eliminates six percent of the executed instructions which create arithmetic expressions and eliminates seven percent of the executed instructions which evaluate these structures and decreases the static code size by ten percent. A modified WAM-database execution model is proposed which reduces static code by twenty percent with a modest four percent increase in executed instructions. We find Prolog to attain only modest speedups through simple pipelined and multiple-operation-issue machine implementations and propose new directions to explore to allow Prolog to better exploit the increasing levels of instruction-level parallelism attainable in modern architectures.

# Table of Contents

# List of Figures

# List of Tables

# Acknowledgements

I would foremost like to thank my advisor Professor Gurindar S. Sohi for his guidance, support and especially his understanding and patience during the years I have been under his supervision. I hope one day to gain the admiration and respect of my students to the degree I admire Guri for the dedication, persistance and the wisdom and values he brings to his profession.

I am grateful to the Univerity of Wisconsin-Madison for the research assistantship they provided to me during the 1989-1990 academic year. Were it not for the persistance of Professor Larry Landweber, it is unlikely I would have received the assistantship and it is *less* likely that I would have completed my degree! I thank Larry for the efforts he put forth in my behalf, his encouragement, his time and attention, and his invaluable advice.

I thank my dissertation committee members: Professors Ken Kunen, Mark Hill, Yannis Ioannidis and John Beetem for their time and helpful comments and Professor Jim Goodman for serving on my preliminary examination committee.

Peter Van Roy and Barry Fagin provided an early version of the PLM compiler and PLM simulator. Without the availability of their tools, inner details of the WAM would have remained mysterious. I am grateful to Van Roy, Fagin and the Aquarius project for providing their compiler and allowing me to incorporate it into my work.

The IBM Corporation, largely through efforts of Robert C. Moran, provided financial support during my first three years in Madison through the IBM Resident Graduate Fellowship Program.

I thank again Professor Jim Goodman and Professor Larry Travis for their support earlier on in the graduate program.

I am grateful to Bob Holloway for selecting me as an instructor and enjoyed our frequent conversations. Teaching 302 was an invaluable experience and teaching has become my career.

To the many housemates, officemates, graduate students, undergraduates, faculty, staff, Madisonians, spouses, significant others, fraternity brothers, volunteers, and those that I've missed that I've had the fortune to meet during

six years in Madison, I say thanks and apologize for not listing each and every one of you. I would like to single out one thank you, not so much for the friendship he provided over the years, but for the request I make now. Marty, since you've worked so hard to organize so many of my affairs in the past, I was hoping you'd generate a list of the individuals referenced above and send them individual letters. *Thanks!*

And to my family and old friends, I should have a little more time for you now . . .

# Table of Contents

**A CHARACTERIZATION OF PROLOG EXECUTION**

Mark A. Friedman

Under the supervision of Associate Professor Gurindar S. Sohi

at the University of Wisconsin-Madison

We analyze the execution of a new suite of medium-sized and realistic benchmarks through simulations on a general-purpose, register-oriented architecture at the abstract machine and architectural levels to identify the most critical characteristics of Prolog for efficient execution. We propose improvements through modest enhancements to the architecture and to the Warren abstract machine to add significant support for the identified issues. Architectural support for tag-handling operations through an architecture which distinguishes between the tag and value fields of an object leads to a twelve percent decrease in static code and a seven percent decrease in executed instructions. The addition of orthogonal tag instructions reduces code size by an additional nine percent and reduces dynamic instructions by eleven percent. The introduction of push and pop instructions decreases code size by seven percent and the number of instructions executed by eight percent. Static code is reduced by forty percent and the number of instructions executed decreases by nine percent by utilizing procedure argument mode information at compile time within compiled unification operations. Knowledge of dereferencing characteristics reduces static code by six percent and reduces instructions executed by six percent by eliminating dereferencing for objects which are directly referenced and when previously dereferenced values may be substituted for non-dereferenced objects. Optimization of arithmetic expression evaluation eliminates six percent of the executed instructions which create arithmetic expressions and eliminates seven percent of the executed instructions which evaluate these structures and decreases the static code size by ten percent. A modified WAM-database execution model is proposed which reduces static code by twenty percent with a modest four percent increase in executed instructions. We find Prolog to attain only modest speedups through simple pipelined and multiple-operation-issue machine implementations and propose new directions to explore to allow Prolog to better exploit the increasing levels of instruction-level parallelism attainable

in modern architectures.