Chapter 1

Introduction

‘‘You’re given the form,
but you have to write the sonnet yourself.
What you say is completely up to you.’’
– Madeleine L’Engle, *A Wrinkle In Time*

1. Thesis statement

The purpose of this dissertation is to provide constructive proof that the logic programming language Prolog can be implemented an order of magnitude more efficiently than the best previous systems, so that its speed approaches imperative languages such as C for a significant class of problems.

The motivation for logic programming is to let programmers describe *what* they want separately from *how* to get it. It is based on the insight that any algorithm consists of two parts: a logical specification (the *logic*) and a description of how to execute this specification (the *control*). This is summarized by Kowalski’s well-known equation Algorithm = Logic + Control [40]. Logic programs are statements describing properties of the desired result, with the control supplied by the underlying system. The hope is that much of the control can be automatically provided by the system, and that what remains is cleanly separated from the logic. The descriptive power of this approach is high and it lends itself well to analysis. This is a step up from programming in imperative languages (like C or Pascal) because the system takes care of low-level details of how to execute the statements.

Many logic languages have been proposed. Of these the most popular is Prolog, which was originally created to solve problems in natural language understanding. It has successful commercial implementations and an active user community. Programming it is well understood and a consensus has developed regarding good programming style. The semantics of Prolog strike a balance between efficient implementation and logical completeness [42, 82]. It attempts to make programming in a subset of first-order logic practical. It is a naive theorem prover but a useful programming language because of its mathematical foundation, its simplicity, and its efficient implementation of the powerful concepts of unification (pattern matching) and search (backtracking).

Prolog is being applied in such diverse areas as expert systems, natural language understanding, theorem proving [57], deductive databases, CAD tool design, and compiler writing [22]. Examples of successful applications are AUNT, a universal netlist translator [59], Chat-80, a natural language query system [81], and diverse in-house expert systems and CAD tools. Grammars based on unification have become popular in natural language analysis [55, 56]. Important work in the area of languages with implicit parallelism is based on variants of Prolog. Our research group has used Prolog successfully in the development of tools for architecture analysis [12, 16, 35], in compilation [19, 73, 76], and in silicon compilation [11].

Prolog was developed in the early 70's by Colmerauer and his associates [38]. This early system was an interpreter. David Warren's work in the late 70's resulted in the first Prolog compiler [80]. The syntax and semantics of this compiler have become the de facto standard in the logic programming community, commonly known as the Edinburgh standard. Warren's later work on Prolog implementation culminated in the development of the Warren Abstract Machine (WAM) in 1983 [82], an execution model that has become a standard for Prolog implementation.

However, these implementations are an order of magnitude slower than imperative languages. As a result, the practical application of logic programming has reached a crossroads. On the one hand, it could degenerate into an interesting academic subculture, with little use in the real world. Or it could flourish as a practical tool. The choice between these two directions depends crucially on improving the execution efficiency. Theoretical and experimental work suggests that this is feasible—that it is possible for an implementation of Prolog to use the powerful features of logic programming only where they are needed.

Therefore I propose the following thesis:

A program written in Prolog can execute as efficiently as its implementation in an imperative language. This relies on the development of four principles:

> **(1) An instruction set suitable for optimization.**
> **(2) Techniques to exploit the determinism in programs.**
> **(3) Techniques to specialize unification.**
> **(4) A global dataflow analysis.**

2. The Aquarius compiler

I have tested this thesis by constructing a new optimizing Prolog compiler, the Aquarius compiler. The design goals of the compiler are (in decreasing order of importance):

(1) **High performance**. Compiled code should execute as fast as possible.

(2) **Portability**. The compiler's output instruction set should be easily retargetable to any sequential architecture.

(3) **Good programming style**. The compiler should be written in Prolog in a modular and declarative style. There are few large Prolog programs that have been written in a declarative style. The compiler will be an addition to that set.

I justify the four principles given in the thesis statement in the light of the compiler design:

(1) **Reduce instruction granularity**. To generate efficient code it is necessary to use an execution model and instruction set that allows extensive optimization. I have designed the Berkeley Abstract Machine (BAM) which retains the good features of the Warren Abstract Machine (WAM) [82], namely the data structures and execution model, but has an instruction set closer to a sequential machine architecture. This makes it easy to optimize BAM code as well as port it to a sequential architecture.

(2) **Exploit determinism**. The majority of predicates written by human programmers are intended to be executed in a deterministic fashion, that is, to give only one solution. These predicates are in effect case statements, yet systems too often compile them inefficiently by using backtracking to simulate conditional branching. It is important to replace backtracking by conditional branching.

(3) **Specialize unification.** Unification is the foundation of Prolog. It is a general pattern-matching operation that can match objects of any size. Its logical semantics correspond to many possible actions in an implementation, including passing parameters, assigning values to variables, allocating memory, and conditional branching. Often only one of these actions is needed, and it is important to simplify the general mechanism. For example, one of the most common actions is assigning a value to a variable, which can often be simplified to a single load or store.

(4) **Dataflow analysis**. A global dataflow analysis supports techniques to exploit determinism and specialize unification by deriving information about the program at compile-time. The BAM instruction set is designed to express the optimizations possible by these techniques.

Simultaneously with the compiler, our research group has developed a new architecture, the VLSI-BAM, and its implementation. The first of several target machines for the compiler is the VLSI-BAM. The interaction between the architecture and compiler design has significantly improved both. This dissertation describes only the Aquarius compiler. A description of the VLSI-BAM and a cost/benefit analysis of its features is given elsewhere [34, 35].

3. Structure of the dissertation

The structure of the dissertation mirrors the structure of the compiler. Figure 1.1 gives an overview of this structure. Chapter 2 summarizes the Prolog language and previous techniques for its high performance execution. Chapters 3 through 6 describe and justify the design of the compiler in depth. Chapter 3 discusses its two internal languages: kernel Prolog, which is close to the source program, and the BAM, which is close to machine code. Chapter 4 gives the optimizing transformations of kernel Prolog. Chapter 5 gives the compilation of kernel Prolog into BAM. Chapter 6 gives the optimizing transformations of BAM code. Chapter 7 does a numerical evaluation of the compiler. It measures its performance on several machines, does an analysis of the effectiveness of its optimizations, and briefly compares its performance with the C language. Finally, chapter 8 gives concluding remarks and suggestions for further work.

The appendices give details about various aspects of the compiler. Appendix A is a user manual for the compiler. Appendices B and C give a formal definition of BAM syntax and semantics. Appendix D is an English description of BAM semantics. Appendix E describes the extended DCG notation, a tool that is used throughout the compiler's implementation. Appendix F lists the source code of the C and Prolog benchmarks. Appendix G lists the source code of the compiler.

Figure 1.1 – Structure of the compiler and the dissertation

4. Contributions

4.1. Demonstration of high performance Prolog execution

A demonstration that the combination of a new abstract machine (the BAM), new compilation techniques, and a global dataflow analysis gives an average speedup of five times over Quintus Prolog [58], a high performance commercial system based on the WAM. This speedup is measured with a set of medium-sized, realistic Prolog programs. For small programs the dataflow analysis does better, resulting in an average speedup of closer to seven times. For programs that use built-in predicates in a realistic manner, the average speedup is about four times, since built-in predicates are a fixed cost. The programs for which dataflow analysis provides sufficient information are competitive in speed with a good C compiler.

On the VLSI-BAM processor, programs compiled with the Aquarius compiler execute in 1/3 the cycles of the PLM [28], a special-purpose architecture implementing the WAM in microcode. Static code size is three times the PLM, which has byte-coded instructions. The WAM was implemented on SPUR, a RISC-like architecture with extensions for Lisp [8], by macro-expansion. Programs compiled with Aquarius execute in 1/7 the cycles of this implementation with 1/4 the code size [34].

4.2. Test of the thesis statement

A test of the thesis that Prolog can execute as efficiently as an imperative language. The results of this test are only partially successful. Performance has been significantly increased over previous Prolog implementations; however the system is competitive with imperative languages only for problems for which dataflow analysis is able to provide sufficient information. This is due to the following factors:

- I have imposed restrictions on the dataflow analysis to make it practical. As programs become larger, these restrictions limit the quality of the results.
- The fragility of Prolog: minor changes in program text often greatly alter the efficiency with which the program executes. This is due to the under-specification of many Prolog programs, i.e. their logical meaning rules out computations but the compiler cannot deduce all cases where this happens.

For example, often a program is deterministic (does not do backtracking) even though the compiler cannot figure it out. This can result in an enormous difference in performance: often the addition of a single cut operation or type declaration reduces the time and space needed by orders of magnitude.

- The creation and modification of large data objects. The compilation of single assignment semantics into destructive assignment (instead of copying) in the implementation, also known as the *copy avoidance problem*, is a special case of the general problem of efficiently representing time in logic. A quick solution is to use nonlogical built-in predicates such as `setarg/3` [63]. A better solution based on dataflow analysis has not yet been implemented.

- Prolog's apparent need for architectural support. A general-purpose architecture favors the implementation of an imperative language. To do a fair comparison between Prolog and an imperative language, one must take the architecture into account. For the VLSI-BAM processor, our research group has analyzed the costs and benefits of one carefully chosen set of architectural extensions. With a 5% increase in chip area there is a 50% increase in Prolog performance.

4.3. Development of a new abstract machine

The development of a new abstract machine for Prolog implementation, the Berkeley Abstract Machine (BAM). This abstract machine allows more optimization and gives a better match to general-purpose architectures. Its execution flow and data structures are similar to the WAM but it contains an instruction set that is much closer to the architecture of a real machine. It has been designed to allow extensive low-level optimization as well as compact encoding of operations that are common in Prolog. The BAM includes simple instructions (register-transfer operations for a tagged architecture), complex instructions (frequently needed complex operations), and embedded information (allows better translation to the assembly language of the target machine). BAM code is designed to be easily ported to general-purpose architectures. It has been ported to several platforms including the VLSI-BAM, the SPARC, the MIPS, and the MC68020.

4.4. Development of the Aquarius compiler

The development of the Aquarius compiler, a compiler for Prolog into BAM. The compiler is sufficiently robust that it is used routinely for large programs. The compiler has the following distinguishing features:

- It is written in a modular and declarative style. Global information is only used to hold information about compiler options and type declarations.
- It represents types as logical formulas and uses a simple form of deduction to propagate information and improve the generated code. This extends the usefulness of dataflow analysis, which derives information about predicates, by propagating this information inside of predicates.
- It is designed to exploit as much as possible the type information given in the input and extended by the dataflow analyzer.
- It incorporates general techniques to generate efficient deterministic code and to encode each occurrence of unification in the simplest possible form.
- It supports a class of simplified unbound variables, called *uninitialized variables*, which are cheaper to create and bind than standard variables.

The compiler development proceeded in parallel with the development of a new Prolog system, Aquarius Prolog [31]. For portability reasons the system is written completely in Prolog and BAM code. The Prolog component is carefully coded to make the most of the optimizations offered by the compiler.

4.5. Development of a global dataflow analyzer

The development of a global dataflow analyzer as an integral part of the compiler. The analyzer has the following properties:

- It uses abstract interpretation on a lattice. Abstract interpretation is a general technique that proceeds by mapping the values of variables in the program to a (possibly finite) set of *descriptions*. Execution of the program over the descriptions completes in finite time and gives information about the execution of the original program.

- It derives a small set of types that lets the compiler simplify common Prolog operations such as variable binding and unification. These types are uninitialized variables, ground terms, nonvariable terms, and recursively dereferenced terms. On a representative set of Prolog programs, the analyzer finds nontrivial types for 56% of predicate arguments: on average 23% are uninitialized (of which one third are passed in registers), 21% are ground, 10% are nonvariables, and 17% are recursively dereferenced. The sum of these numbers is greater than 56% because arguments can have multiple types.
- It provides a significant improvement in performance, reduction in static code size, and reduction in the Prolog-specific operations of trailing and dereferencing. On a representative set of Prolog programs, analysis reduces execution time by 18% and code size by 43%. Dereferencing is reduced from 11% to 9% of execution time and trailing is reduced from 2.3% to 1.3% of execution time.
- It is limited in several ways to make it practical. Its type domain is small, so it is not able to derive many useful types. It has no explicit representation for aliasing, which occurs when two terms have variables in common. This simplifies implementation of the analysis, but sacrifices potentially useful information.

4.6. Development of a tool for applicative programming

The development of a language extension to Prolog to simplify the implementation of large applicative programs (Appendix E). The extension generalizes Prolog’s Definite Clause Grammar (DCG) notation to allow programming with multiple named accumulators. A preprocessor has been written and used extensively in the implementation of the compiler.