The Making of the Aquarius Prolog System

Peter Van Roy

Computer Science Division
University of California, Berkeley
vanroy@ernie.berkeley.edu

Alvin M. Despain

Department of Electrical Engineering – Systems
University of Southern California
despain@cse.usc.edu

1. Introduction

As part of the Aquarius project at Berkeley we have developed a high performance Prolog system. The performance of the system has reached a milestone: it can run some programs more efficiently than the MIPS C compiler doing its best optimization. In this paper we present performance results and we discuss informally some practical aspects of the implementation techniques that achieved these results. This paper is a companion to the conference paper [7]. For more information see [1, 6].

We discuss practical issues such as the tools used in the compiler's implementation, the interaction with the VLSI-BAM architecture design, the implementation of high-performance built-in predicates, and the efficient interface of Prolog with assembly language. We also discuss the limitations of the system and suggestions for future research.

2. Some performance numbers

The table below compares the performance of Aquarius Prolog with MIPS C on a 25 MHz MIPS processor. Measurements are given for tak, fib, and hanoi, which are recursion-intensive integer functions, and for quicksort, which sorts a 50 element list 10,000 times. Prolog and C code is available by anonymous ftp to arpa.berkeley.edu. In all cases the user time is measured with the Unix ‘‘time’’ utility. We have been careful to encode the same algorithm for Prolog and C, in a natural style for each. The C versions are compiled both with no optimization and with the optimization level that produces the fastest code (usually level 4). The Prolog versions are compiled with the Aquarius compiler's global dataflow analysis.

This comparison is not exhaustive—there are so many factors involved (quality of the implementations, choice of programs/algorithms to compare, programming style, architectural support) that we do not attempt to address this issue in its entirety. We intend only to dispel the notion that implementations of Prolog are inherently slow because of its expressive power. We demonstrate that it is possible for a Prolog compiler to simplify the expressive power of Prolog to the level of an imperative language. What is remarkable is that the logical semantics of the language are unchanged.

The two quicksort implementations are careful to use the same pivot elements. The C implementation uses an array of integers and does in-place sorting. The Prolog implementation uses lists and creates a new sorted list. The list representation uses two words for each data element. Coincidentally, the Prolog version is twice as slow as the C version, the same as the ratio of the data sizes.

Comparing Prolog and C (in seconds)			
Benchmark	Aquarius	MIPS C	
	Prolog	Unoptimized	Optimized
tak(24,16,8)	1.2	2.1	1.6
fib(30)	1.5	2.0	1.6
hanoi(20,1,2,3)	1.3	1.6	1.5
quicksort	2.8	3.3	1.4

We have analyzed the cause of the compiler's good performance for these benchmarks. There are four reasons:

(1) Determinism. The Aquarius compiler can deduce that these functions are deterministic, so that no backtracking overhead is incurred.

(2) Environments (stack frames) are allocated per clause in Prolog, whereas they are allocated for the whole procedure in C. For the functions in the table, the Prolog version can often return without having to create an environment.

(3) Outputs are returned in registers (dataflow analysis determines that this is safe). This optimization is performed by both Prolog and C.

(4) Last call optimization converts recursion into iteration. This optimization allows functions with a single recursive call to execute with constant stack space. This is essential for Prolog because recursion is its main looping construct. The C compiler does not do last call optimization. C has constructs to denote iteration explicitly (e.g. ''for'' and ''while'' loops) so it does not need this optimization as badly as Prolog does.

3. Practical issues in the compiler's design and implementation

3.1. Software engineering

The implementation of the compiler was guided by the quality of the code generated during the development process. The simplest methods that gave the required results were chosen for each optimization. High performance is not something that appears full blown, but it is implemented piecemeal, as a large collection of small optimizations. We start with a set of well-written benchmarks of various sizes and remove performance bottlenecks one by one as they appear. The optimizations are ranked according to (estimated benefits) / (estimated implementation effort), and the best ones (e.g. uninitialized variables, determinism, and dataflow analysis) are implemented first. This approach was felt to result in the most effective use of the available development time, which was about three man years. The resulting compiler is optimized for a good programming style.

The compiler was developed using C-Prolog and Quintus Prolog. C-Prolog was used for daily development and debugging because it allows more rapid turnaround when programs are modified. Quintus Prolog was used for the final product because it is faster and can run large programs. These two systems have almost completely identical behavior. The problems we found porting between the two are minor and syntactic.

3.2. Portability of the system

The run-time system implements a superset of the built-in predicates of C-Prolog [1, 4]. For portability reasons it is written completely in Prolog and BAM assembly. It consists of about

6,500 lines of Prolog and about 1,000 lines of BAM assembly code. The Prolog component is carefully coded to make the most of the optimizations offered by the compiler and thus to reduce the performance penalty for implementing built-in predicates in Prolog. Calls to built-ins are compiled using fast entry points—depending on the argument types known at the call, the built-in is replaced by the most specific entry point available. The BAM component is efficiently interfaced to Prolog using fast parameter passing mechanisms based on uninitialized variable types. A macro definition facility further improves performance.

The first target machine for the system is the VLSI-BAM processor [2]. The BAM was developed concurrently with the VLSI-BAM processor, but the two instruction sets are quite different. The VLSI-BAM is constrained by its hardware implementation; the BAM evolved by looking at the requirements of Prolog and is designed to allow a great deal of low-level optimization. BAM code is easily macro-expanded into the instruction set of the VLSI-BAM as well as general-purpose processors.

We intend to retarget the system to several popular machines including MC68020, MIPS, and SPARC, and for this purpose our group is building a portable back-end to the compiler. Porting the system requires more than just macro-expanding the BAM code, as we must also port the system libraries and Prolog built-ins. We have recently rewritten the back-end, libraries, and built-ins to simplify the task of porting the complete system. Porting to a new machine requires writing only a small machine-dependent module.

3.3. Programming in an applicative style

As far as possible, the compiler was written in a purely applicative style, i.e. by passing information locally in arguments and not using side-effects. Global data is used only for keeping information about predicate modes and compiler options. However, in practice the number of arguments becomes large, which makes writing and maintaining programs difficult. For the Aquarius compiler we have developed a preprocessor which extends Prolog with an arbitrary number of arguments without increasing the size of the source code [5]. The extended Prolog is a generalization of the Definite Clause Grammar (DCG) notation [3]. The preprocessor implements an unlimited number of named accumulators, and handles all the tedium of parameter passing. If written explicitly, each accumulator requires two additional arguments. As a result, the bulk of the program source does not depend on the number of accumulators, so maintaining and extending it is simplified. For single accumulators the notation defaults to the standard DCG notation.

4. The future

The main thrust of this work has been to reduce the performance gap between Prolog and imperative languages. However, the goal of achieving parity with imperative languages has been achieved only for the class of programs that do not copy large objects and for which dataflow analysis can provide sufficient information. To further improve performance we need to address these problems. We observe that when writing a program, a programmer commonly has a definite intention about the data's type (intending predicates to be called only in certain ways) and about the data's lifetime (intending data to be used only for a limited period). Because of this consistency, we postulate that a dataflow analyzer should be able to derive this information and a compiler should be able to take advantage of it.

There has been much good theoretical work on global analysis for Prolog, but few implementations, and fewer still that are part of a system that takes advantage of the information. Our work shows that a simple dataflow analysis scheme integrated into a compiler is already quite useful. However, we have imposed many restrictions on its generality to make the implementation practical. As programs become larger, these restrictions limit the quality of the results. We hope the success of this experiment encourages others to relax these restrictions. For example, it would not be too difficult to:

- Extend the domain to represent common types such as integers, proper lists, and nested compound terms. In large programs it seems to be important to look inside of compound terms.
- Extend the domain to represent variable aliasing explicitly. This avoids the loss of information that affects our analyzer.
- Extend the domain to represent data lifetimes. This is useful to replace copying of compound terms by in-place destructive assignment. The term ‘‘compile-time garbage collection’’ that has been used to describe this process is a misnomer; what is desired is not just memory recovery, but to preserve as much as possible of the old value of the compound term. Often a new compound term similar to the old one is created at the same time the old one becomes inaccessible. Destructive assignment is used to modify only those parts that are changed.
- Extend the domain to represent types for each invocation of a predicate. For example, the analyzer could keep track not only of argument types for predicate definitions, but of argument types for goals inside the definitions. This is useful to implement multiple specialization, i.e. to make separate copies of a predicate called in several places with different types.

To guide the implementation of these improvements it is important to build large applications and study the interaction between programming style and the effectiveness of analysis. This will lead ultimately to a successor to Prolog that is both closer to logic and efficiently implementable.

5. Concluding remarks

We conclude that the execution speed of Prolog has been significantly improved. It is an encouraging sign for the future of logic programming. However, much remains to be done before logic languages are competitive with imperative languages for all programs. We hope that our success will encourage others to continue this work.

6. Acknowledgements

We wish to thank all the members of the BAM project, in particular Tep Dobry and Bruce Holmer for VLSI-BAM architecture design and software development, Ralph Haygood for the built-ins, and Mike Carlton for the cache simulator and the universal translator. The preliminary MIPS port was done by Mike and Bruce. This work is partially funded by the Defense Advanced Research Projects Agency (DARPA) and monitored by the Office of Naval Research under contract No. N00014-88-K-0579. Access to a MIPS processor has been graciously provided by the Mammoth project under National Science Foundation Infrastructure Grant number CDA-8722788.

7. References

1. R. Haygood, *Aquarius Prolog User Manual and Aquarius Prolog Implementation Manual*, University of California, Berkeley, 1990 (to appear).
2. B. K. Holmer, B. Sano, M. Carlton, P. Van Roy, R. Haygood, J. M. Pendleton, T. Dobry, W. R. Bush and A. M. Despain, Fast Prolog with an Extended General Purpose Architecture, *17th International Symposium on Computer Architecture*, May 1990, pp. 282-291.
3. F. C. N. Pereira and D. H. D. Warren, Definite Clause Grammars for Language Analysis—A Survey of the Formalism and a Comparison with Augmented Transition Networks,

International Journal of Artificial Intelligence Vol. 13, 3 (May 1980), pp. 231-278, North-Holland.

4. F. Pereira, ed., *C-Prolog User’s Manual – Version 1.5,* University of Edinburgh, Dept. of Architecture, February 1984.

5. P. Van Roy, A Useful Extension to Prolog’s Definite Clause Grammar notation, *SIGPLAN Notices Vol. 24*, 11 (November 1989), pp. 132-134.

6. P. Van Roy, Can Logic Programming Execute as Efficiently as Imperative Programming?, *Ph.D. Thesis*, Expected December 1990.

7. P. Van Roy and A. M. Despain, The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler, *North American Conference on Logic Programming ’90,* October 1990.