Chapter 7

Evaluation of the Aquarius system

1. Introduction

This chapter attempts to quantify some of the ideas that were introduced in previous chapters. The evaluation process is as important as any other part of the implementation of a large software system. During the design phase it guides the design decisions. After the design is complete, it shows what features of the design contributed most to its effectiveness and it gives a foundation for starting the next design. Quantitative measurements are the most reliable guideposts one has during the design. For example, it is easy to imagine many possible compiler optimizations, but most of these have an insignificant effect on performance. It is more difficult to discover optimizations that are widely applicable.

Five evaluations are performed in this chapter:

(1) The absolute performance of the system.

(2) The effectiveness of the dataflow analysis.

(3) The effectiveness of the determinism transformation.

(4) A brief comparison with a high performance implementation of the C language.

(5) A bug analysis, summarizing the number and types of bugs encountered during development.

Table 7.1 describes the benchmarks used in this chapter and their size in lines of code (not including comments). The benchmarks were chosen as examples of realistic programs doing computations representative of Prolog. This includes benchmarks that spend much of their time executing built-in predicates because this behavior is common in real-world programs. The benchmarks are divided into two classes, *small* and *large*, depending on whether the compiled code with analysis is smaller or larger than 1000 words. The benchmarks log10, ops8, times10, and divide10 are grouped together and referred to as *deriv* because they are closely related. The benchmarks are available by anonymous ftp to arpa.berkeley.edu.

All VLSI-BAM numbers in this chapter were obtained from the VLSI-BAM instruction-level simulator and include cache effects [17]. The simulated system has 128 KB instruction and data caches. The

Table 7.1 – The benchmarks

Benchmark	Lines	Description
nreverse	10	Naive reverse of a 30-element list.
tak	15	Recursive integer arithmetic.
qsort	19	Quicksort of a 50-element list.
log10	27	Symbolic differentiation.
ops8	27	Symbolic differentiation.
times10	27	Symbolic differentiation.
divide10	27	Symbolic differentiation.
serialise	29	Calculate serial numbers of a list.
queens_8	31	Solve the eight queens puzzle.
mu	33	Prove a theorem of Hofstadter's ''mu-math.''
zebra	36	A logical puzzle based on constraints.
sendmore	43	The SEND+MORE=MONEY puzzle.
fast_mu	54	An optimized version of the mu-math prover.
query	68	Query a static database (with integer arithmetic).
poly_10	86	Symbolically raise a polynomial to the tenth power.
crypt	64	Solve a simple cryptarithmetic puzzle.
meta_qsort	74	A meta-interpreter running qsort.
prover	81	A simple theorem prover.
browse	92	Build and query a database.
unify	125	A compiler code generator for unification.
flatten	158	Source transformation to remove disjunctions.
sdda	273	A dataflow analyzer that represents aliasing.
reducer_nowrite	298	A graph reducer based on combinators.
reducer	301	Same as above but writes its answer.
boyer	377	An extract from a Boyer-Moore theorem prover.
simple_analyzer	443	A dataflow analyzer analyzing qsort.
nand	493	A logic synthesis program based on heuristic search.
chat_parser	1138	Parse a set of English sentences.
chat	4801	Natural language query of a geographical database.

caches are direct mapped and use a write-back policy. They are run in warm start: each benchmark is run twice and the results of the first run are ignored. The cache overhead is greatest for tak compiled without analysis, and for poly_10, simple_analyzer, chat, and boyer. For these programs it ranges from 9% to 24%. For meta_qsort, reducer, and chat_parser the overhead ranges from 2% to 3%. For all other programs the overhead is less than 0.5%.

2. Absolute performance

This section compares the performance of Aquarius Prolog with Quintus Prolog. Tables 7.2 and 7.3 compare the performance of Quintus Prolog version 2.5 running on a Sun 4/65 (25 MHz SPARC) with that of Aquarius Prolog running on the VLSI-BAM (30 MHz). The ''Raw Speedup'' column gives the ratio of the speeds. The ''Normalized Speedup'' column divides this ratio by 1.8. Our group is in the process of

porting the Aquarius system to the MIPS, MC68020, and SPARC processors. It was not possible to get numbers for these systems in time for the final version of this dissertation.

The normalization factor of 1.8 takes into account the Prolog-specific extensions of the VLSI-BAM (a factor of 1.5) and the clock ratio (a factor of 30/25 = 1.2). The general-purpose base architecture of the VLSI-BAM is very similar to the SPARC. The effect of the architectural extensions of the VLSI-BAM [34] has been carefully measured to be about 1.5 for large programs. However, for the small programs the compiler is able to remove many Prolog-specific features, so that the normalized speedup numbers in Table 7.2 are an underestimate.

Table 7.2 – Performance results for small programs (in ms)

Benchmark	Size (lines)	Quintus v2.5 (Sun 4/65)	Aquarius (VLSI-BAM)	Normalized Speedup	Raw Speedup
deriv		1.143	0.0913	7.0	12.5
log10	27	0.153	0.0168		
ops8	27	0.239	0.0189		
times10	27	0.345	0.0257		
divide10	27	0.406	0.0299		
nreverse	10	1.62	0.136	6.6	11.9
qsort	19	4.820	0.173	15.5	27.8
serialise	29	3.10	0.447	3.9	6.9
query	68	23.7	3.57	3.7	6.6
mu	33	7.04	0.808	4.8	8.7
fast_mu	54	9.08	0.932	5.4	9.7
queens_8	31	21.2	1.13	10.4	18.7
tak	15	1120.	25.4	24.5	44.1
poly_10	86	417.	35.5	6.5	11.7
sendmore	43	490.	38.4	7.1	12.8
zebra	36	423.	84.1	2.8	5.0
geometric mean				6.7	12.1
standard deviation of mean				1.9	3.3

For the small benchmarks, the normalized speedup is somewhere between 6.7 and 12.1 (Table 7.2). The normalized speedup of the large benchmarks without built-in predicates is about 5.2 (Table 7.3). Speedup is better for the small benchmarks because dataflow analysis is able to derive better types for many of them. For some of them (such as tak and nreverse) it derives essentially perfect types. The small programs show a large variation in speedups. The tak benchmark does well because it relies on integer arithmetic, which is compiled efficiently using uninitialized register types. The zebra benchmark does poorly for two reasons. First, it does a large amount of backtracking, which is inherently limited by memory bandwidth. Second, it works by successively instantiating arguments of a compound data

Table 7.3 – Performance results for large programs (in ms)					
Benchmark	Size (lines)	Quintus v2.5 (Sun 4/65)	Aquarius (VLSI-BAM)	Normalized Speedup	Raw Speedup
No built-ins					
prover	81	8.67	0.921	5.2	9.4
meta_qsort	74	49.6	4.71	5.8	10.5
nand	493	173.3	13.7	7.0	12.7
reducer_nowrite	298	312.	37.2	4.6	8.4
chat_parser	1138	1157.	129.5	5.0	8.9
browse	92	5450.	741.	4.1	7.4
geometric mean				5.2	9.4
standard deviation of mean				0.5	0.8
Including built-ins					
unify	125	18.3	1.40	7.2	13.0
flatten	158	13.6	1.42	5.3	9.6
sdda	273	29.5	2.94	5.6	10.0
crypt	64	21.7	4.00	3.0	5.4
simple_analyzer	443	180.	33.4	3.0	5.4
reducer	301	405.	44.9	5.0	9.0
chat	4801	3100.	699.	2.5	4.4
boyer	377	4870.	1360.	2.0	3.6
geometric mean				3.8	6.9
standard deviation of mean				0.7	1.3
geometric mean (all large programs)				4.4	7.9

Table 7.4 – Time spent in built-in predicates		
Benchmark	Time (%)	Most used built-ins
prover	0	–
meta_qsort	0	–
chat_parser	0	–
nand	<1	–
browse	1	length/2
reducer	40	write/1, compare/3, arg/3
unify	40	arg/3, functor/3, compare/3
crypt	50	div/2, mod/2, */2
boyer	60	arg/3, functor/3
simple_analyzer	70	compare/3, sort/2, arg/3
sdda	70	write/1, =../2, compare/3
flatten	80	write/1, sort/2, compare/3, name/2, functor/3, arg/3

structure. The analysis algorithm does not have a representation for this operation, so it cannot be optimized.

The built-in predicates in Aquarius Prolog are not greatly faster than those in Quintus Prolog, since many of the Quintus built-ins are not written in Prolog, but in hand-crafted assembly. The Aquarius system shows better speedup over Quintus built-ins written in Prolog (such as `read/1` and `write/1`) and the entry specialization transformation also speeds up the built-ins. Table 7.4 gives the percentage of time that

the benchmarks spend executing inside built-in predicates. This number does not take into account built-ins that are implemented as in-line code (arithmetic test, addition and subtraction, and type checking). The table also gives the most often used built-in predicates for each benchmark in decreasing order of usage.

Several benchmarks use built-in predicates significantly. The normalized speedup for these programs is 3.8, somewhat less than programs without built-ins (Table 7.3). The normalized speedup for all large programs is 4.4 (the reducer benchmark is counted only once in this average). The boyer benchmark does poorly because it relies heavily on the `arg/3` and `functor/3` built-in predicates. The chat benchmark uses these built-ins as well as others including `setof/3`, but it was not possible to measure the fraction of execution time spent in them. The sdda and flatten benchmarks do well partly because the `write/1` built-in is much faster in Aquarius than in Quintus.

3. The effectiveness of the dataflow analysis

This section evaluates the effectiveness of the dataflow analysis with three kinds of measurements. Tables 7.5, 7.6, and 7.7 give the effect of the dataflow analyzer on performance and code size, and the efficiency of the analyzer both in terms of its execution time and the fraction of arguments for which types can be deduced.

For a representative set of realistic Prolog programs of various sizes up to 1,100 lines, the analyzer is able to derive type information for 56% of all predicate arguments. It finds that on average 23% of all predicate arguments are uninitialized, 21% of arguments are ground, 10% of arguments are nonvariables, and 17% of arguments are recursively dereferenced. The sum of these three numbers is greater than 56% since it is possible for an argument to have multiple types, e.g. it can be ground and recursively dereferenced at the same time. Doing analysis reduces execution time on the VLSI-BAM by 18% for programs without built-ins and static code size by 43% for all programs.

Table 7.5 gives the execution time in microseconds of the benchmarks for the VLSI-BAM compiled without analysis (No Modes) and with analysis (Auto Modes). The last three columns give the ratios of the auto modes to the no modes times. To give an idea how built-ins affect the results of analysis, Table 7.5 gives two performance ratios for the large benchmarks: the first for all programs, and the second for

Table 7.5 – The effect of dataflow analysis on performance

Benchmark	No Modes (μs)			Auto Modes (μs)			Auto/No Modes		
	Time	Deref	Trail	Time	Deref	Trail	Time	Deref	Trail
deriv	146	18.2	5.5	91.3	0.3	0.1	0.63	0.02	0.02
log10	25.9	2.3	0.7	16.8	0	0			
ops8	28.5	3.3	1.0	18.9	0.3	0.1			
times10	39.7	5.1	1.3	25.7	0	0			
divide10	51.7	7.5	2.5	29.9	0	0			
nreverse	308	79.7	31.1	136	0	0	0.44	0.00	0.00
qsort	378	109	25.1	173	0	0	0.46	0.00	0.00
serialise	512	75.8	12.3	447	44.9	0.7	0.87	0.59	0.05
mu	992	154	48.0	783	139	34.7	0.79	0.90	0.72
fast_mu	1120	148	38.0	932	64.4	7.9	0.83	0.44	0.21
queens_8	1700	271	67.9	1090	33.4	0	0.64	0.12	0.00
query	5180	560	174	3570	0	0	0.69	0.00	0.00
tak	71700	13800	3180	25400	0	0	0.35	0.00	0.00
poly_10	60400	6280	1740	35600	1080	209	0.59	0.17	0.12
zebra	84600	11400	8.6	84100	11400	8.4	0.99	1.00	0.98
average							0.66	0.29	0.19
prover	1070	110	29.4	820	51.2	5.9	0.76	0.47	0.20
unify	1600	198	33.9	1400	138	19.3	0.88	0.69	0.57
flatten	1460	149	9.9	1420	133	6.5	0.97	0.90	0.66
sdda	3180	368	36.9	2940	296	21.3	0.92	0.81	0.58
crypt	4090	319	104	4000	262	104	0.98	0.82	1.00
meta_qsort	5330	674	182	4450	417	63.0	0.83	0.62	0.35
nand	18700	2290	542	13400	902	22.9	0.72	0.39	0.04
simple_analyzer	35400	3880	316	31900	3080	76.2	0.90	0.79	0.24
reducer	48800	6680	1210	44900	5580	731	0.92	0.84	0.61
chat_parser	151000	19400	6990	131000	11200	4360	0.87	0.58	0.62
browse	820000	117000	28600	741000	96700	20400	0.90	0.82	0.71
boyer	1410000	73900	6340	1360000	75000	6270	0.97	1.02	0.99
average							0.89	0.73	0.55
average (no built-ins)							0.82	0.58	0.39

programs that do not use built-ins significantly (the first five of Table 7.4). Data initialization times are subtracted from deriv, nreverse, qsort, serialise, and prover. The table also gives the time each benchmark spends performing dereferencing (Deref) and trailing (Trail).

The time spent in dereferencing and trailing, two of the most common Prolog-specific operations, is significantly reduced by analysis. For the small benchmarks analysis reduces dereferencing from 17% to 5% of execution time, and trailing from 4% to 0.6% of execution time. This is because they are simple enough that analysis is able to deduce most relevant modes. For the large benchmarks dereferencing is reduced from 11% to 9% and trailing is reduced from 2.3% to 1.3%. These results are less extreme for two reasons: the large benchmarks use built-ins, which are unaffected by analysis, and the analyzer loses infor-

Table 7.6 – The effect of dataflow analysis on static code size

Benchmark	No Modes (instructions)	Auto Modes (instructions)	Auto/No Modes
tak	80	34	0.42
nreverse	287	139	0.48
queens_8	472	146	0.31
qsort	485	215	0.44
deriv	5891	1123	0.19
log10	1464	272	
ops8	1469	277	
times10	1479	287	
divide10	1479	287	
query	1425	403	0.28
serialise	860	520	0.60
mu	1169	731	0.63
fast_mu	1165	718	0.62
zebra	1271	814	0.64
poly_10	3023	893	0.30
average			0.45
crypt	1239	1027	0.83
browse	1863	1150	0.62
prover	4395	1318	0.30
meta_qsort	2484	1424	0.57
flatten	4267	2335	0.55
unify	6326	4210	0.67
sdda	6526	5031	0.77
simple_analyzer	9057	5836	0.64
nand	23406	6654	0.28
reducer	11726	7682	0.66
boyer	24862	9136	0.37
chat_parser	33557	20516	0.61
average			0.57

mation due to its inability to handle aliasing and its limited type domain.

Table 7.6 gives the static code size (in VLSI-BAM instructions) for the benchmarks compiled without analysis (No Modes) and with analysis (Auto Modes). The effect of analysis on code size is greater than the effect on performance. This follows from the compiler's implementation of argument selection: when no modes are given, the compiler generates more code to handle arguments of different types. If analysis derives the type then the code becomes much smaller. The code size compares favorably with other symbolic processors, and is low enough that there is no disadvantage to having a simple instruction set. With the analyzer, code size on the VLSI-BAM is similar to the KCM [6], about three times the PLM, a micro-coded WAM [28], and about one fourth the SPUR using macro-expanded WAM [8].

Table 7.7 – The efficiency of dataflow analysis

Benchmark	Args	Preds	Time (sec)	Modes (fraction of arguments) uninit	ground	nonvar	rderef	any
deriv	12	8	11.9	0.33	0.67	0.00	0.67	1.00
log10	3	2	2.9					
ops8	3	2	3.0					
times10	3	2	3.0					
divide10	3	2	2.9					
tak	4	2	2.3	0.25	0.75	0.00	0.75	1.00
nreverse	5	3	2.2	0.40	0.60	0.00	0.60	1.00
qsort	7	3	3.4	0.43	0.57	0.00	0.57	1.00
query	7	5	4.2	0.86	0.14	0.00	0.14	1.00
zebra	10	6	3.5	0.10	0.00	0.50	0.00	0.60
serialise	16	7	4.2	0.38	0.19	0.06	0.19	0.63
queens_8	16	7	6.0	0.31	0.69	0.00	0.69	1.00
mu	17	8	9.6	0.12	0.47	0.00	0.12	0.65
poly_10	27	11	16	0.33	0.67	0.00	0.67	1.00
fast_mu	35	7	21	0.29	0.55	0.05	0.55	0.89
average				0.35	0.48	0.06	0.45	0.89
meta_qsort	10	7	11	0.30	0.00	0.10	0.00	0.40
crypt	18	9	12	0.00	0.61	0.11	0.56	0.72
prover	22	9	13	0.27	0.09	0.27	0.14	0.68
browse	42	14	20	0.24	0.45	0.05	0.40	0.74
boyer	62	25	31	0.27	0.00	0.06	0.00	0.34
flatten	83	28	34	0.27	0.08	0.16	0.11	0.52
sdda	87	32	45	0.18	0.07	0.17	0.08	0.44
reducer	134	41	50	0.13	0.10	0.05	0.12	0.29
unify	141	29	84	0.18	0.19	0.14	0.21	0.56
nand	180	43	5900	0.26	0.67	0.00	0.28	0.93
simple_analyzer	270	71	77	0.23	0.10	0.08	0.10	0.41
chat_parser	744	156	263	0.44	0.19	0.02	0.09	0.67
average				0.23	0.21	0.10	0.17	0.56

Table 7.7 presents data about the efficiency of the dataflow analyzer. For each benchmark it gives the number of predicate arguments (Args) where a predicate of arity N is counted as N, the number of predicates (Preds), the analysis time (Time), the fraction of arguments that are uninitialized (uninit), ground (ground), nonvariable (nonvar), or recursively dereferenced (rderef), and the fraction of arguments that have any of these types (any). Analysis time is measured under Quintus release 2.0 on a Sun 3/60. It is roughly proportional to the number of arguments in the program, except for the nand benchmark. The sum of the individual modes columns is usually greater than the any modes column. This is because arguments can have multiple modes—they can be both recursively dereferenced and ground or nonvariable. Uninitialized arguments are present in great quantities, even in large programs such as chat_parser and simple_analyzer. Comparing the small and large benchmarks, the fraction of derived modes decreases for

the large programs for each type except nonvariable. For both the small and large benchmarks the analyzer transforms one third of the uninitialized modes into uninitialized register modes.

4. The effectiveness of the determinism transformation

To show what parts of the determinism transformation of Chapter 4 are the most effective, it is useful to define a spectrum of determinism extraction algorithms ranging from pure WAM to the full mechanism of the Aquarius compiler. To do this, the Aquarius mechanism for extracting determinism is divided into three orthogonal axes:

(1) The kind of tests used to extract determinism. These tests are separated into three classes: explicit unifications (e.g. `X=a`, `X=s(Y)`), arithmetic tests (e.g. `X<Y`, `X>1`), and type checks (e.g. `var(X)`, `atomic(X)`). Pure WAM uses only explicit unifications with nonvariables. Aquarius uses all three kinds.

(2) Which argument(s) are used to extract determinism. Pure WAM uses only the first argument of a predicate. Aquarius uses any argument that it can determine is effective. It uses enrichment heuristic 2 (Chapter 4 section 6.2).

(3) Whether the factoring transformation is performed (Chapter 4). Factoring significantly increases determinism for predicates that contain many identical compound terms in the head. Pure WAM does not assume factoring. Aquarius does factoring by default.

These three parameters define a three-dimensional space of determinism extraction algorithms. Each algorithm is characterized by a 3-tuple depending on its position on each of the axes (Table 7.8). This results in $3 \times 2 \times 2 = 12$ data points. Pure WAM selection corresponds to the first element in each column, denoted by the 3-tuple (U, ONE, NF). The Aquarius compiler's selection corresponds to the last element in each column, denoted by the 3-tuple (UAT, ANY, F).

For each of these 12 points three parameters were measured: execution time, static code size, and compile time. All programs are compiled with dataflow analysis and executed on the VLSI-BAM. All averages are geometric means. It was only possible to do measurements for nine benchmarks: nreverse, qsort, query, mu, fast_mu, queens_8, flatten, meta_qsort, and nand. Therefore the variance of the results is large and

Table 7.8 – Three dimensions of determinism extraction		
Kind of test	Which argument	Factoring
Explicit unifications only (*U*).	First argument only (*ONE*).	No factoring (*NF*).
Explicit unifications and arithmetic tests (*UA*).	Any argument (*ANY*).	Do factoring (*F*).
Explicit unifications, arithmetic tests, and type checks (*UAT*).		

they can be relied upon only to indicate trends. The benchmarks were written for the WAM. The measurements compare only the relative powers of different kinds of determinism extraction in the BAM. They do not compare the WAM and BAM directly.

Figure 7.1 – The effectiveness of determinism extraction

Figure 7.1 depicts the 12 points as a lattice. Each vertex denotes one particular combination of determinism extraction. The top element corresponds to Aquarius selection and the bottom element corresponds to WAM selection. Each edge connects two points that differ by one step in one coordinate. The vertices

are marked with the percent slowdown compared to Aquarius selection. The edges are marked with the percent difference in execution time between their two endpoints.

The mean speedup for the nine benchmarks when going from WAM selection (U, ONE, NF) to Aquarius selection (UAT, ANY, F) is 16%. There is no significant change in mean code size for any of the twelve data points. The variance of the compile time is too large to make any conclusions about it.

The mean speedup of factoring is 8%. However, factoring is the only transformation that sometimes slows down execution. The factoring heuristic should be refined to look inside compound arguments to see whether there is any potential determinism there. If there is none, it should not factor that argument.

One way of finding a set of effective extensions for determinism extraction is by traversing the lattice from bottom to top, and picking the edge with the greatest performance increase at each vertex. Starting at WAM selection (U, ONE, NF), the first extension is the ability to use arithmetic tests in selection. This speeds up execution by 3%. The second extension is the ability to select on any argument. This speeds up execution by another 3%. The third extension is the factoring transformation. This speeds up execution by 8%. At this point, the resulting performance is within 2% of Aquarius selection.

The resulting vertex (UA, ANY, F) seems to be a particularly good one, i.e. the ability to select on arithmetic tests in any argument works well together with factoring. Leaving out any one of these three extensions reduces performance by at least 8%. A plausible reason for this result is that the benchmarks do many arithmetic tests on the arguments of compound terms and it is only the combination of the three extensions that is able to compile this deterministically.

5. Prolog and C

The performance of Aquarius Prolog is significantly better than previous Prolog systems. A question one can pose is how the system compares with an implementation of an imperative language. This section presents a comparison of Prolog and the C language on several small programs. The comparison is not exhaustive—there are so many factors involved that I do not attempt to address this issue in its entirety. I intend only to dispel the notion that implementations of Prolog are inherently slow because of its expressive power. A serious comparison of two languages requires answering the following questions:

(1) How can implementations of different languages be compared fairly? This comparison concentrates exclusively on the language and ignores features external to the language itself, such as user interface, development time, and debugging abilities. One method is to pick problems to be solved, and then to write the ‘‘best’’ programs in each language to solve the problems, choosing the algorithms appropriate for each language. The disadvantages of this approach are (a) different languages are appropriate for different problems, (b) how does one decide when one has written the ‘‘best’’ program? To avoid these problems I have chosen to compare algorithms, not programs.

(2) Which algorithms will be implemented in both languages? Ideally one should select a range of algorithms, from those most suited to imperative computations (e.g. array computations) to those most suited to symbolic computation (e.g. large dynamic data objects, pattern matching). Prolog is at an advantage at the symbolic end of the spectrum because to implement symbolic computations in an imperative language we effectively have to implement more and more of a Prolog-like system in that language. The programmer does the work of a compiler. At the imperative end of the spectrum, the efficiency of Prolog depends strongly on the ability of the compiler to simplify its general features.

(3) What programming style will be used in coding the algorithms? I have made an attempt to program in a style which is acceptable for both languages. This includes choosing data types in both languages that are natural for each language. For example, in Prolog dynamic data accessed by pointers is easiest to express, whereas in C static arrays are easiest to express. It is possible to use dynamic data in C, but it requires more effort and is used only for those tasks that need it specifically.

(4) How are architectural features taken into account? For fairness both implementations should run on the same machine. The measurements use the same processor, the MIPS, for both implementations. However, a general-purpose architecture favors the execution of imperative languages, since it has been designed to execute such languages well. This shows up for algorithms whose Prolog implementation makes heavy use of Prolog-specific features. To allow the reader to make an informed judgment, the table does not correct for this fact. It is important to bear in mind that by adding additional architectural features comprising 5% of the chip area to the VLSI-BAM (a pipelined processor similar in many ways to the MIPS), the performance increases by 50% for programs that use

Prolog-specific features (compiled with the current version of the Aquarius compiler). Architectural studies done by our research group suggest that these features could be added to a future MIPS processor.

Table 7.9 compares the execution time of small algorithms coded in both C and Prolog 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 10000 times. Prolog and C source code is available by anonymous ftp to arpa.berkeley.edu. In all cases the user time is measured with the Unix ‘‘time’’ utility. The C versions are compiled with the standard MIPS C compiler using both no optimization and the optimization level that produces the fastest code (usually level 4). The Prolog versions are compiled with dataflow analysis and translated into MIPS assembly by a partial translator. The same algorithms were encoded for both Prolog and C, in a natural style for each. The natural style in C is to use static data, whereas in Prolog all data is allocated dynamically.

Table 7.9 – Comparing Prolog and C (in sec)

Benchmark	Aquarius Prolog	MIPS C	
		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

Recursive functions are fast in Prolog for three reasons: last call optimization converts recursion into iteration, environments (stack frames) are allocated per clause and not per procedure as in C, and outputs are returned in registers (they are of uninitialized register type). Last call optimization allows functions with a single recursive call to execute with constant stack space. This is essential for Prolog because recursion is its only looping construct. The MIPS 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 strongly. The time for `fib(30)`, the only recursive integer function that is not able to use last call optimization in Prolog, is closest to C.

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 needs two words to store each data element. Coincidentally, the

Prolog version is twice as slow as the C version, the same as the ratio of the data sizes.

Table 7.10 – Classification of bug types

Kind	Description	%
Mistake	A part of the compiler that is incorrect due to an oversight. When many mistakes occur related to one particular area, then they become hotspot bugs.	39
• Local	A problem that can be fixed by changing just a few predicates. For example, it may be due to a typographical error or a simple oversight in a predicate definition.	(37)
• Global	A problem that can be fixed only with many changes throughout the compiler. This kind of mistake is more fundamental. For example, avoiding the generation of BAM instructions with double indirections requires many small changes.	(3)
Incomplete	A part of the compiler whose first implementation is incomplete because of incomplete understanding of its purpose. Later use stretches it beyond what it was intended to do, so that it needs to be extended and/or cleaned up. For example, the updating of type formulas when new information is given.	19
Hotspot	A critical area of the compiler that requires much thinking to get correct. Its importance is much greater than its size would indicate. Such an area gets more than its share of mistakes.	16
• Conceptual	A concept in the compiler design whose implementation is prone to many mistakes. For example, the concept of uninitialized variables.	(13)
• Physical	A part of the compiler's text. For example, symbolic unification in the dataflow analyzer and parameter passing in the clause compiler both resulted in many bugs.	(14)
Mixture	An undesired interaction between separate parts of the compiler. Despite careful design, often the separate transformations and optimizations are not completely orthogonal, but interact in some (usually limited) way. For example, maintaining consistency between the dataflow analyzer and the clause compiler. This leads to the dereference chain transformation, which in its turn leads to the problem of interaction between it and the preferred register allocation.	16
Improvement	A possible improvement in the compiler. This is not strictly a bug, but it may point to an important optimization that could be added to the compiler. For example, a possible code optimization or reduction in compilation time.	9
Understanding	A problem due to the programmer misunderstanding the required input to the compiler. This is not strictly a bug, but it may point to difficulties in the compiler's user interface or in the language. For example, the difference between the terms `_is_` and `_<_` in Prolog. The first is a variable and the second is a structure.	4

6. Bug analysis

This section gives an overview of the number and types of bugs encountered during compiler development. A *bug* in a program is a problem that leads to incorrect or undesired behavior of the program. In the compiler, this means incorrect or slow compilation, or slow execution of compiled code.

Table 7.10 classifies the bugs found during development [76]. (The percentages do not add up to 100% because bugs can be of more than one type.)

The development of the compiler started early 1988 and proceeded until late 1990. An extensive suite of test programs was maintained to validate versions of the compiler. The test suite was continually extended with programs that resulted in bugs and with programs from external sources. Records were kept of all bugs reported by users of the compiler other than the developer. A total of 79 bug reports were sent from January 1989 to August 1990 by five users. The frequency of bug reports stayed constant near four per month. Statistical analysis is consistent with the distribution being random with no time dependence, i.e. the number of bug reports fluctuates, but there is no increasing or decreasing trend. Therefore the development introduced bugs at about the same rate as they were reported and fixed. This coincidence can be explained by postulating that the time spent developing was limited by the necessity of having to spend time debugging to maintain a minimum level of robustness in the compiler. This is consistent with my personal experience during the development process.