Scalable Acceleration of Inductive Logic Programs

Andreas Fidjeland, Wayne Luk and Stephen Muggleton
Department of Computing
Imperial College
180 Queen's Gate
London SW7 2BZ, England

Abstract

Inductive logic programming systems are recognised as an emerging but powerful paradigm for machine learning which can make use of background knowledge to produce theories expressed in logic. They have been applied successfully to a wide range of problem domains, from protein structure prediction to satellite fault diagnosis. However, their execution can be computationally demanding. We introduce a scalable FPGA-based architecture for executing inductive logic programs, such that the execution speed largely increases linearly with respect to the number of processors. The architecture contains multiple processors derived from the Warren Abstract Machine, which has been optimised for hardware implementation using techniques such as instruction grouping and speculative assignment. The effectiveness of the architecture is demonstrated using the mutagenesis data set containing 12000 facts of chemical compounds.

1 Introduction

Inductive Logic Programming (ILP) is a relatively new tool in the arsenal of the scientist. It combines machine learning with logic programming, and can produce first-order logic theories based on examples. A strength of ILP systems as compared with other machine learning systems is that ILP makes use of background knowledge and can therefore build on partial theories within a field. ILP also produces theories in a form that is human-readable and can therefore be used in the normal scientific discourse.

ILP systems have produced new knowledge of use to experts within a domain, such as rules for prediction of the activity of untried drugs. In [1] the ILP system Golem is shown to outperform other learning systems in predicting the secondary structure of proteins. This has been one of the hardest open problems in molecular biology and is of great interest to the pharmaceutical industry. The problem is to predict the placement of the main three dimensional sub-structures of the protein, given a sequence of amino acid residues. Golem has an accuracy of around 80% while previous learners had an accuracy of between 50 and 60%. Other problem domains which have successfully been treated by ILP systems include learning diagnosis rules for satellites [2], creating innovative designs from first principles [3] and learning finite element mesh analysis design rules [4]. The induction process is computationally demanding and can run for hours.

This paper demonstrates that for the execution of ILP programs, the performance of an FPGA-based coprocessor scales well with respect to chip size. Progol, the ILP system studied here, makes a large number of calls to a Prolog interpreter, with calls independent of each other. Several Prolog implementations can be mapped onto the same chip. The speed with which problems are solved is proportional to the number of Prolog implementations, and this, in turn, grows with the size of the chip.

The paper outlines the design of an FPGA-system based on the Warren's Abstract Machine (WAM) execution model for Prolog. The main achievements are the following:

1. Adaption of the WAM for hardware using simplified term representation.

2. Optimisation of the hardware-based WAM by fine-grained parallelisation within instructions by means of speculative assignments and assignment grouping.

3. An architecture containing several WAMs on the same chip. The architecture is general in that it can be applied to a range of applications facing a large number of similar independent sub-problems. However, the architecture fits the nature of ILP problems well.

4. The designs are compared to the software-based version, using a large data set. The scalability of the multi-WAM architecture is evaluated.

This paper is organised as follows. In section 2 an overview is given of ILP . In the sections 3 and 4 the single-WAM and multiple-WAM architectures are described. In section 5 an evaluation of the above architectures are given. Finally, section 6 contains a few concluding remarks.

2 Background

In the field of artificial intelligence, systems often have to acquire knowledge which can, for one reason or another, not be directly coded into the system. Machine learning comprises a number of techniques, such as neural networks and belief networks, aimed at this knowledge acquisition. Inductive logic programming (ILP) [5] is another technique, combining machine learning with logic programming. ILP systems produce predicate descriptions from background information and examples.

ILP systems aim to find the simplest consistent hypothesis which can explain the given background information and examples. In the ILP system Progol [6], the learning process is viewed as a search problem in the space of potential hypothesis. Each example is generalised with respect to the background knowledge. This is done by finding the most specific hypothesis explaining the example and searching for an optimal hypothesis which lies between the most specific and most general hypothesis in terms of generality. This search space forms a sub-lattice bounded above by the most general hypothesis and below by the least general hypothesis.

Progol uses an A*-like algorithm to find the maximally compressive hypothesis, where compression takes into account the ability of the hypothesis to explain the example set as well as the complexity of the hypothesis. The search starts with the most general hypothesis and moves through the search space guided by compression. In the worst case all the possible hypothesis must be considered. For each hypothesis which is considered, Progol establishes whether or not each example is a consequence of it. Herein lies the computational complexity, as a large number of candidate hypothesis must be considered, and for each of these there can be a large number of examples to test. When the optimal generalisation of an example is found, it is added to the background knowledge. The background knowledge may then explain some of the examples which have yet to be generalised, and these are removed from the example set. Progol then proceeds by generalising the remaining examples in turn.

More formally we want

$$B \wedge H \models E$$

where B is the background knowledge, H is the hypothesis and E is the example set. E can contain both positive and negative examples; it states both what is known to be true and what is known to be false. B, H, and E can be arbitrary logic programs, essentially databases of facts and rules. The most common logic programming formalism is Prolog which is based on clausal form logic and resolution. A program consists of facts (‘$g(x).$’) and rules (‘$f(x)$ *if* $g(x).$’), while a computation determines whether a query (‘$f(x)?$’) is a consequence of a program.

It can be shown that if $\overline{\bot}$ is the conjunction of ground literals which are true in all models of $B \wedge \overline{E}$ and H and E are restricted to single Horn clauses, then the following holds:

$$B \wedge E \models \overline{\bot} \models \overline{H}$$

and so

$$H \models \bot$$

where $\bot$ is the most specific hypothesis.

The complete set of H are those clauses which imply $\bot$. Progol searches for H among the clauses which θ-subsumes $\bot$. A clause c_1 θ-subsumes another clause c_2 if and only if there exists a substitution θ such that $c_1\theta \subseteq c_2$, i.e. c_1 is more general than c_2. The hypothesis search space forms a lattice such that $\Box \preceq H \preceq \bot$, where $\preceq$ denotes θ-subsumption and $\Box$ is the most general hypothesis.

As an example of an ILP program, take the mutagenesis data set, which is used as a benchmark below. The data set describes a set of compounds, classified as either “active” or “inactive” with respect to mutagenicity, i.e. their ability to alter DNA. The background knowledge consists of structural definitions of the compounds. Examples consists of facts describing compounds as either active or inactive. The hypothesis generated by Progol are rules defining compounds as active if they haver certain structural properties.

Ohwada and Mizoguchi have presented two approaches for using hardware to speed up Progol. The first [8] makes use of parallelism on three levels. First it induces concepts in parallel, if there are more than one. Secondly it executes branches in the hypothesis search in parallel. Finally it counts positive and negative examples in parallel. Our approach makes use of the latter form of parallelism, although the architecture is different. The second approach [9] uses logic programming to solve goals and concurrent logic programming to dispatch goals to machines. This ILP engine is distributed over several processors with the hypothesis search task allocated dynamically. The performance scales well with the number of processors.

3 Single WAM Design

The Progol coprocessor is based on the WAM, Warren’s Abstract Machine [10]. The WAM is an execution model for Prolog which has become its *de facto* standard implementation technique. It is a stack-based architecture with an instruction set corresponding closely to Prolog code. The instruction set is small, but the instructions are quite complex.

Like imperative machines, the WAM has instructions for sequential control but also for unification and backtracking. The local stack (STACK) holds environments (activation records) for local variables in addition to choice points keeping information needed to reset the state of the machine upon backtracking.

Term data are typed dynamically; data items can change type at run-time through unification. During unification variables can be bound, and these bindings are kept in a separate binding stack (BIND, not present in the original WAM). A separate stack, the TRAIL, records (*trails*) the bindings which must be undone (*detrailed*) upon backtracking. In addition the WAM has a memory containing the code (CODE) and a table (WCODE) mapping predicates (such as $f(x)$ above) to their corresponding code.

The WAM uses indexing on the first argument of a predicate to reduce the number of clauses that must be unified with when calling a predicate. The code for clauses defining a predicate are grouped together according to the type and value of the first argument. Another two instructions pass control to the correct code section using another table (HTAB) which maps constants to code sections.

Our first WAM design for FPGAs, the S-WAM, is a sequential adaption of a simplified WAM written for Progol. It is a 32-bit architecture for compatability with the Progol host. The S-WAM uses six types of terms: integer, skolem constant (e.g. a), character string, variable, floating point number and predicate (e.g. $f(a,b)$). The terms are identified with a 3-bit tag indicating their type, while the remaining 29-bits are used to hold the value. For integers the value is the integer itself. For skolem constants and strings the value is an index into a Progol table. For variables the value is a small integer denoting the offset into the associated binding frame where its binding is recorded. Floats require more than 32 bits so the value field is a pointer to the floating point value itself. The value for the predicate is a pointer to the main body of the predicate. This consists of the predicate name, its arity and all the argument terms.

Instructions have a fixed width of 32 bits with a 5-bit opcode. The fixed format has been chosen to minimise code fetch and decoding overheads. There are 15 instructions, each with a 5-bit opcode. The redundant opcode bit is kept for currently unsupported instructions for cuts and variable calls.

The organisation of the memory is guided by imperative studies of the access frequencies and size requirements for the various segments. The memory access frequencies below, are the percentage of the total number of memory accesses found in the mutagenesis benchmark. Memory segments are kept either on-chip in distributed RAM, in on-chip block RAM and in off-chip RAM.

- The TERM segment, containing all the clauses in the logic program, is large. The segment is read-only and frequently accessed (40%). Because of the size of this segment we have chosen to keep it off-chip, but it could be kept in on-chip block RAM if there is enough space. The term data for our mutagenesis benchmark are too large (440KB) to keep on the XCV2000E chip we used, but it could fit on in the memory of more recent chips.

- The CODE segment contains the code for the clauses in TERM. Since WAM-code corresponds closely to Prolog-code, the two segments are of roughly equal size. This segment is read-only but is less frequently accessed (8%) than TERM. It is therefore kept off-chip.

- WCODE maps each defined predicate in TERM into its code section in CODE. The minimum size for WCODE is equal to the number of predicates. Clearly this segment need only be a fraction of the size of TERM. WCODE is only accessed when a predicate is called and this happens infrequently (0.3%). It is therefore kept off-chip.

- The HTAB segment used for indexing contains entries for distinct constants in first argument position of a head of a rule. This can be larger than WCODE since each set of clauses defining a predicate may have several such constants. The segment is only accessed infrequently (1%) and is therefore kept off-chip.

- BIND contains binding frames stacked chronologically. The size of this segment is proportional to the depth of execution. The execution depth is bounded by Progol so this segment is quite small. BIND is accessed when setting up a binding frame, binding variables, trailing and detrailing bindings and when dereferencing variables. This happens frequently (26%) so BIND should be kept in on-chip block RAM.

- Likewise, STACK, containing environment and choice point frames, is also stacked chronologically. Effectively, STACK consists of two interleaved stacks. The size of STACK is proportional to execution depth and is therefore of limited size, like BIND. Accesses to STACK are frequent (23%) and it should be kept in on-chip block RAM.

- TRAIL contains pointer to a subset of the bindings found on the binding stack. This segment is therefore smaller again than BIND. Trailing and detrailing happens very infrequently (1%), but because of the small size of the segment it should be kept in on-chip block RAM.

- The push-down-list (PDL) is a stack used during unification. This need only be of limited size. It is used only by unify so this stack is kept local to that instruction in distributed RAM.

The S-WAM is optimised using fine-grained parallelisation within each instruction. This is done by grouping assignments together and by making speculative assignments to reduce the cycle count for each instruction. The resulting design, the P-WAM, is still a sequential control machine, but with improved performance. The effect of this low-level parallelisation varies between the instructions. In general, memory accesses is the limiting factor of the parallelisation, and instructions spending much time processing data rather than reading and writing data, shows better improvement.

- The unification instruction has a good performance benefit for all types of unifications, as much time is spent extracting data fields and determining the types of data. For example unifying the two terms *f(X,b,c)* and *f(a,Y,Z)*, where X, Y and Z are unbound variables, is reduced from 189 to 90 cycles.

- The control flow instructions have their cycle counts halved.

- Stack instructions saving and restoring information on the run-time stack are memory intensive. Because of sequential stack accesses the speedup of these instructions is minimal.

- For the two indexing instructions the cycle count is halved.

- The five ancillary operations called by the WAM instructions vary in the degree to which they can be parallelised. Three of them are so small that parallelisation has little effect. The most important one, for dereferencing, gain very little from parallelisation because the instruction is memory intensive looping through bindings and terms.

Parallelising instructions means that segments may have to be accessed in parallel. This can be done at two points. The first is fetching code in parallel with instruction execution requiring CODE to be accessed in parallel with the other segments. The effect of this parallel access is estimated to be a 4% speedup. The second point is by trailing unwinding the trail upon backtracking while setting the values in the choice point frame. The effect of this is also 4%. In our implementation all memory accesses are sequential.

4 Multiple WAM Design

The P-WAM is small enough for several to be placed on the same chip. Fortunately ILP problems have an inherent parallelism which can take advantage of an architecture with several processors. The test for covering for hypothesised clauses requires that a large number of clauses are tested against the set of examples, each test being an independent call to Prolog.

The M-WAM (figure 1) contains a single controller communicating with the Progol host and a series of P-WAM processors. The Progol host places background knowledge (CODE, TERM, WCODE and HTAB) in memory accessibly both to the FPGA chip and the host. When queries for testing covering are created during hypothesis search, the code and term data associated with them is places in the shared memory. The queries themselves (pointers into CODE) are then passed to the controller which dispatches them to the available processors.

The controller and the processors run as independent threads communicating over channels. The controller spawns off the processor threads and then waits for the Progol host to pass a query through the control register. When a query is received, the controller monitors the available channels to the processors and dispatches the query as soon as a channel opens. The individual processors busy-wait for a request from the controller. When a processors receives a request, it serves it and waits for another one.

The P-WAM processors have their own run-time data structures (STACK, TRAIL, BIND and PDL). Like in the single-WAM designs, the three first of these are kept in on-chip block RAM, while the PDL is kept in distributed RAM. Since accesses to CODE are rare, this segment can be shared between several processors, with a memory arbiter keeping several processors from accessing the segment at the same time. For the P-WAM 4% of cycles are spent fetching code. TERM is far more frequently accessed so sharing is a bigger problem. This segment should therefore be replicated as much as possible. If there is room in on-chip RAM it could be kept there as the large number of small blocks makes sharing easier. We choose to keep it off-chip, but replicated in each of the available RAM blocks.

The M-WAM contains a different number of processors depending on the size of the target chip. In order to sim-

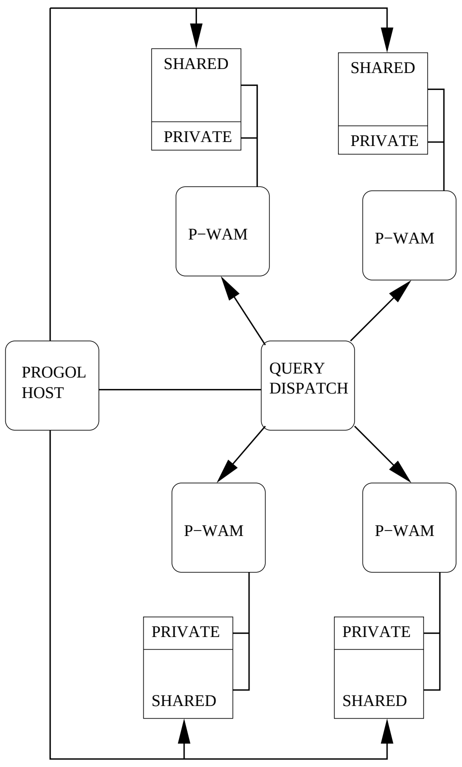

Figure 1: M-WAM architecture showing the channels for query dispatching and buses to the various memories. Shared memory is accessible to both the host and the M-WAM, while private memory are the run-time structures of each processors in the M-WAM.

plify the process of creating code for different targets a generator is used (figure 2). The code for the M-WAM can be generated given two inputs: memory layout and number of processors. The memory layout and interface must be specified for each chip. The memory description file defines the memory structure of the target system: size and number of blocks of both off- and on-chip memory. It must also define the size and placement of each segment. The full M-WAM description is created in three stages:

1. Combine memory description with P-WAM description. Shared data accesses are set to point to the correct block.

2. Replicate the combined P-WAM design. Definitions private to the P-WAMs (functions and data) must be kept in separate name spaces.

3. Expand the M-WAM header. This file contains the controller. The channel communication section must be expanded to contain the correct number and names of channels. The header is then combined with the rest of the code.

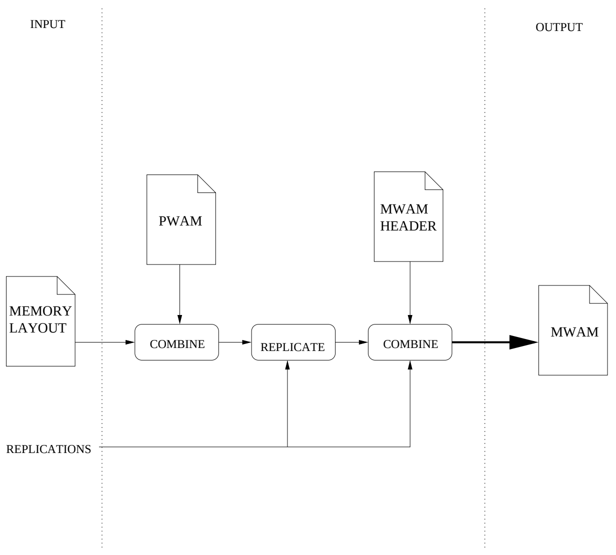

Figure 2: M-WAM generator. Creates an M-WAM description given the number of required P-WAMs as well as a memory description.

5 Evaluation

The designs are implemented on an XCV2000E chip mounted on an RC1000-PP board and can be clocked at 35MHz. The hardware implementations of the S-WAM and P-WAM are tested using a benchmark based on the mutagenesis data set mentioned above. The benchmark uses the nine rules generated by Progol in [7]. These rules are used as queries with examples as arguments to the various WAM implementations with the background knowledge loaded, i.e. each rule is tested to see whether or not it explains each of the examples correctly. This is done by Progol with each generated candidate hypothesis. The benchmark is not a full run since it tests only the final rules, but is indicative of the performance since the example testing forms the performance bottleneck of the system.

The test returns the execution time for each query. The tested machines are not optimised with respect to the target device, so better performance should be attainable. The software is timed on two different machines. The first is a Pentium III 450MHz with 256MB RAM. The second is a Pentium IV 1.8GHz with 512 MB RAM. The amount of memory is of little significance, since the benchmark programs fits comfortably within the 8MB available on the RC1000-PP.

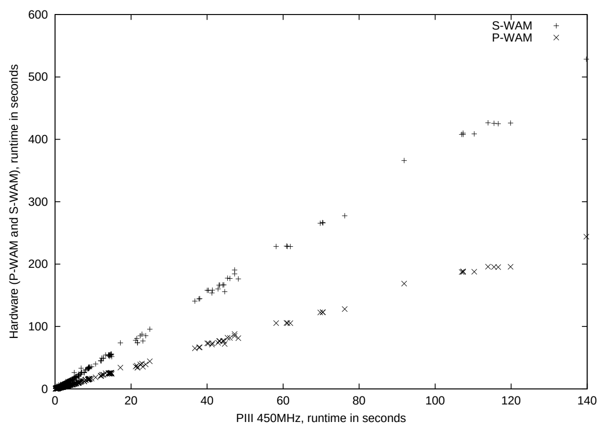

Figure 3: Timing comparison for S-WAM and P-WAM versus the PIII. Each point shows the running time for a query both for hardware (S-WAM or P-WAM) and software. Each query is thus plotted twice. Speedup ratio of hardware to software is constant and the P-WAM is about twice as fast as the S-WAM.

Machine	PIII	PIV
S-WAM	0.43	0.13
P-WAM	0.95	0.27
M-WAM(2)	1.90	0.54
M-WAM(3)	2.84	0.80
M-WAM(4)	3.77	1.07

Table 1: Performance of S-WAM, P-WAM and M-WAM compared to software running on the PIII and PIV. The numbers indicate the speedup factor of the hardware implementation compared to the software implementation. M-WAM(n) means an M-WAM with n P-WAM processors.

The software is timed for each query. The execution times are given for 10000 repeated calls to the same query, as a single query executes too quickly to be detected by the system diagnostics procedures. The S-WAM and P-WAM are also timed for each query, while the M-WAM implementations are timed for the whole benchmark and the speedup is found relative to the P-WAM.

The timing results for each query from software and hardware runs for the S- and P-WAM are plotted in figure 3. The speedup against software is uniform across the queries in the benchmark, as can be seen from the plot where all the points for each machine lie on a straight line. The P-WAM executes queries at about the same rate as the software implementation on the Pentium III, but about 3.5 times slower than on the Pentium IV. The P-WAM executes queries about twice as fast as the S-WAM. The aggregate results are shown in table 1.

Machine	Slices	Increase
S-WAM	3176	1.09
P-WAM	2910	1
M-WAM(2)	5776	1.99
M-WAM(3)	8635	2.97
M-WAM(4)	11479	3.95

Table 2: Space usage for S-WAM, P-WAM and M-WAMs. The increase column displays the size relative to the P-WAM. M-WAM(n) indicates and M-WAM with n P-WAM processors.

Our implementation of the M-WAM with up to four processors shows performance increasing near-linearly (table 1). Like the other designs, the M-WAM is implemented on the XCV2000E chip on the RC1000-PP board with four banks of SRAM. A simplified memory is used with each processor using a separate block of the off-chip memory, so no on-chip block RAM is used. The clock speed the different versions are much the same around 35MHz.

The S-WAM and P-WAM each uses roughly 15% of the XCV2000E chip, which contains 19200 slices. The S-WAM uses 3176 slices, while the P-WAM uses 2910. The reason the P-WAM is smaller than the S-WAM is that it has been optimised more. The space usage of M-WAMs increases near-linearly with the number of P-WAMs used for up to four P-WAMs (table 2).

There are limitations on the speedup that can be achieved by executing processes in parallel. For a set of problems, the upper bound on the speedup is the speedup attained by having as many processors as problems, in which case the execution time is the same as that of the longest problem. The maximum speedup can also be attained with a smaller number of processors, if several problems can be solved in the time it takes to solve the longest one. The maximum speedup is the ratio of total execution time to execution time of the longest problem. The maximum attainable speedup for a given number of processors will tend to increase with the number of problems, since if the added problems are shorter than the longest one, the ratio of total to longest execution time will increase. It follows that for a given number of problems the performance benefit of adding processors yields diminishing returns.

The mutagenesis benchmark consists of nine disjunct sets of problems, one for each rule. The total execution time in a sequential execution is the sum of the execution times of all the queries. The maximum speedup is achieved when each of the nine rules is solved in the time it takes to solve the longest query for that rule. The maximum speedup factor is then the total execution time divided by the sum of the longest runs for the nine rules. This speedup

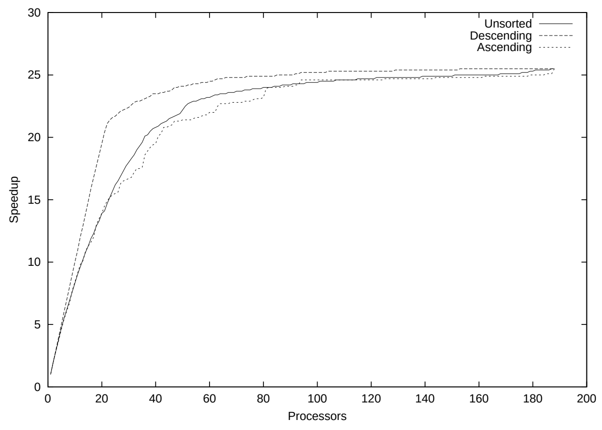

Figure 4: Upper bound on the speedup of M-WAM for increasing number of P-WAMs using the mutagenesis benchmark. The three plots are for queries sorted in descending and ascending order of execution time as well as unsorted. Data are taken from simulation and overheads of having a large number of processors are not taken into account.

factor turns out to be 26.

Maximal speedup can be attained also when there are fewer processors than problems. This is achieved when each processors is equally well utilised, essentially a bin packing problem. Dealing with problems in order of decreasing execution time will keep utilisation even, although not necessarily optimally so. Dealing with problems in order of increasing execution time, on the other hand, will keep it uneven. Figure 4 shows the speedup for the mutagenesis benchmark for up to 188 processors. The simulation uses the cycles counts for each query found for the P-WAM. The three lines shows the speedup compared to a single processor implementation for problems sorted in ascending and descending order of execution time as well as a 'natural' run where the queries are solved in the order given by Progol. The two measurements for sorted queries are interesting in that they indicate the range of speedup that can be attained for a given problem, although to actually sort the queries cannot be done as the execution times are not known.

6 Concluding remarks

This paper demonstrates the feasibility of optimising the ILP system Progol using an FPGA-based Prolog coprocessor. The initial design, the S-WAM, has been improved to produce the P-WAM. An architecture containing multiple P-WAMs has been developed to form the M-WAM, which is capable of running a large number of queries simultaneously. To recapitulate the main points of the design:

1. The S-WAM is an adaption of the WAM for use in FPGA hardware. Terms are simplified and the data are tagged. The various memory segments used by S-WAM have been distributed in the different types of memory available to a typical FPGA-system. This memory structure is based on the access frequency as well as segment sizes.

2. The P-WAM is an optimised version of S-WAM. Instructions are parallelised by means of speculative assignments and assignment grouping. This type of optimisation doubles the speed of the P-WAM with no additional space requirements.

3. The M-WAM combines several P-WAMs together on the same chip. The architecture is general in that it can be applied to a range of applications facing a large number of similar independent sub-problems. However, the architecture fits the nature of ILP problems well.

The above designs have been evaluated using the mutagenesis data set. The main advantage of the resulting design is *scalability*. The M-WAM architecture scales well with the number of processors. The number of processors in turn depends on the chip-size, which increases rapidly. Thus the performance gap, where extra chip-space is left un-utilised, is greatly reduced. Future chips can be filled with a greater number of processors to deal with ever larger problems.

Current and future work consists of the following. First, integrate the implementation fully with Progol, with the compiler targeting the memory available to the FPGA chip. Second, explore device-specific and platform-specific optimisations to improve performance. Third, investigate how the control complexity increases with a larger number of processors. Fourth, investigate the extent to which memory sharing will become a bottleneck for a large number of processors, in particular in the use of on-chip block RAM for TERM data. Finally, explore the use of multiple FPGA-board in order to increase performance while reducing memory requirements by splitting up the examples to be tested into disjunct sets.

Acknowledgements. Many thanks to Shay Ping Seng for his comments and assistance. The support of Xilinx, Inc., Celoxica Limited and UK Engineering and Physical Sciences Research Council (Grant number GR/N 66599) is gratefully acknowledged.

References

[1] R. King, S. Muggleton, R. Lewis, and M. Sternberg. Drug design by machine learning: The use of inductive logic programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductas. *Proceedings of the National Academy of Sciences*, 89(23):11322-11326, 1992.

[2] C. Feng. Inducing Temporal Fault Diagnostic Rules from a Qualitative Model. *Proceedings Eighth International Workshop on Machine Learning*, pp 403 - 406, Morgan Kaufmann, San Mateo, C.A., 1991.

[3] I. Bratko. Innovative design as learning from examples. *Proceedings of the International Conference on Design to Manufacture in Modern Industries*, Bled, Slovenia, June 1993.

[4] I. Bratko and S. Muggleton. Applications of Inductive Logic Programming. *Communications of the ACM*, 38(11):65-70, 1995.

[5] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19,20:629-679, 1994.

[6] S. Muggleton. Inverse Entailment and Progol. *New Generation Computing*, 13:245-286, 1995.

[7] A. Srinivasan, S. Muggleton, R. King, and M. Sternberg. Mutagenesis: ILP experiments in a non-determinate biological domain. In S. Wrobel, editor, *Proceedings of the Fourth International Inductive Logic Programming Workshop*. Gesellschaft für Mathematik und Datenverarbeitung MBH, 1994. GMD-Studien Nr 237.

[8] H. Ohwada and F. Mizoguchi. Parallel Execution for Speeding Up Inductive Logic Programming Systems, *Proc. of the Second International Conference on Discovery Science*, pp. 277-286, 1999.

[9] H. Ohwada, H. Nishiyama, F. Mizoguchi. Concurrent execution of optimal hypothesis search for inverse entailment. *Proc. 10th International Conference on Inductive Logic Programming*, pages 165-173, July 2000.

[10] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, CA, October 1983.