Customising Parallelism and Caching for Machine Learning

Andreas Fidjeland and Wayne Luk
Department of Computing
Imperial College London
{*akf99,wl*}*@doc.ic.ac.uk*

Abstract

Inductive logic programming is an attractive and expressive paradigm for machine learning. A drawback of inductive logic programs is their demanding computational requirements. We present an FPGA-based multi-processor architecture aimed at fast execution of such programs. The architecture exploits both coarse-grained parallelism at the query level, and fine-grained parallelism in the unification algorithm. Instructions are not required, and the components are customised for a hypothesis space referring only to ground unit clauses in the background knowledge. It also benefits from a distributed memory hierarchy, with a method for including background knowledge to eliminate instructions. The effectiveness of this architecture is demonstrated using a large organic chemistry data set. The proposed architecture is faster and smaller than our previous design based on multiple instruction processors. A single customised processor at 38MHz can run 9 times faster than a Pentium 4 processor at 1.8GHz; a Xilinx XCV2000E device can accommodate 24 processors running in parallel.

1 Introduction

Inductive logic programming [1] (ILP) is a successful learning paradigm, with applications in a wide range of fields. ILP systems produce theories in first-order logic, and can also incorporate background knowledge in this form. Based on logic programming, it is highly expressive. However, this expressiveness can have a detrimental impact on performance, and methods for speeding up sequential implementations of ILP are desirable.

This paper describes an architecture for high-performance execution of inductive logic programs in the Progol system. Our main achievements are the following:

- A hardware datapath architecture for a simple and common form of inductive logic programs.
- Exploitation of parallelism at multiple levels. A multi-processor architecture exploits coarse-grained parallelism inherent in Progol, while individual processors exploit data parallelism in unification.
- A cache system for parallel logic processors. The caches are specialised for the input data. A two-level memory hierarchy is used with memories at the highest level shared between processors.
- Evaluation of the architecture on a real data set. We evaluate the benefits of exploiting caching, the effect of different levels of parallelism, and the consequences of changing architectural parameters.

The rest of the paper is organised as follows. Section 2 introduces the problem domain. Section 3 discusses the types of parallelism we exploit. Section 4 presents the processing elements we use in our architecture. Section 5 discusses the cache architecture. Section 6 presents the intended design flow for the architecture. Section 7 presents results and evaluation. Finally Section 8 offers some concluding remarks.

2 Problem Domain

Progol [2] is an instance of inductive logic programming systems. It produces theories in first-order logic based on domain-specific background knowledge and a set of positive and negative examples. Machine learning systems tend to have a trade-off between expressiveness and efficiency. Progol and inductive logic programming lies on the expressive and computationally expensive end of the scale.

The input to Progol consists of background knowledge, a set of examples, and user constraints on the hypothesis space. Based on this, the Progol system constructs a hypothesis space, and employs an A$*$-like search through it. The search heuristic is based on how well the hypothesis under consideration explains the given examples. The output of the system is a list of the best hypothesis and the associated measures of their quality.

active(A) ← *atm(A,B,c,195,C).* (rule 1)

active(A) ← *atm(A,B,c,10,C),*
atm(A,D,c,22,E),
bond(A,D,B,1). (rule 2)

active(A) ← *atm(A,B,c,27,C),*
bond(A,D,E,1),
bond(A,D,B,7). (rule 3)

Figure 1 Example of a hypothesis generated from the mutagenesis data set.

Much of the computational complexity is due to hypothesis testing. The hypothesis space is potentially large, and each hypothesis needs to be tested against each of the examples. Additionally, in order to minimise the degree of overfitting, cross-fold validation is used, which increases the execution time further. Execution time can be tens of hours on conventional sequential machines.

Progol uses the logic programming language Prolog as its description language. The examples, background knowledge and the constructed hypothesis all come in this form. The example tests are queries to a Prolog processor.

Progol and other ILP systems have found many successful applications in molecular biology, such as learning rules for prediction of protein folding [3], mutagenesis [4], and pharmacophore discovery [5]. The background knowledge in this domain commonly consists of long lists of ground facts, such as lists of properties of compounds. We target our acceleration hardware to this type of application domain. The simple form of the background knowledge enables us to do without the full generality of Prolog, so that we can build processors specialised for the hypothesis space.

The rules generated by Progol tend to display a high level of temporal and spatial locality. Testing the rules takes the form of a nested iteration over parts of the background knowledge. As an example, consider the hypothesis in Figure 1 which contains some of the rules produced by Progol when learning to predict mutagenesis in nitroaromatic compounds; highly mutagenic versions of such compounds are believed to be capable of causing cancer.

Rule 3 from Figure 1 states that a compound *A* is mutagenically active if it contains a carbon atom *B* of type 27, that there are two atoms *D* and *E* connected by a bond of type 1, and that *D* is connected to B through a bond of type 7. The rule refers to the predicates *atm/5* and *bond/4* (here 5 and 4 refers to the number of arguments, or arity, of *atm* and *bond*), which are defined by lists of more than 6000 facts each.

Rules, such as the ones given above, are tested against examples such as *active(d18)* which is part of the input to Progol. With Prolog semantics, testing whether a hypothesis holds for rule 3 amounts to first iterating over the facts defining *atm/5* until the constraints are met. Then for each such atom iterate over the facts defining *bond/4* until a suitable bond is found, and finally iterating over *bond/4* again until the second bond is found, at which point the search succeeds. If at one level no solution is found, computation backtracks by finding an alternative solution at the previous level. If no more solutions can be found, the search fails. As one would expect, the execution time is dominated by the innermost loop, in our example the last call to *bond/4*.

In order to speed up the search through a large set of examples, indexing is used, usually based on the first argument. For the *atm/5* and the *bond/4* predicates, this partitions the data set into disjunct subsets for the different compounds. Note that for the rule 3 example the two calls to *bond/4* refer to the same such subset.

Some other parallel architectures for ILP exist. In [6] Ohwada et. al present an ILP engine where the search is distributed dynamically over several processors. The performance scales well for up to 10 processors on their benchmarks, which have a small set of background knowledge, but a large hypothesis space. In [7] Skillicorn and Wang present a parallel version of Progol, with the whole data set partitioned among processors. Processors generate hypothesis based on their own subset, while testing hypothesis based on the entire subset. In contrast we focus on parallelising hypothesis testing, while the hypothesis generation is handled by a host machine. They report linear or super-linear speedup on their benchmarks running on 4 or 6 processor shared-memory machines.

3 Exploiting Parallelism

This section covers two forms of parallelism: query-level parallelism and unification parallelism.

Query-level parallelism, a coarse form of parallelism, is inherent in the Progol algorithm. At each node in the search through the hypothesis space, the hypothesis under consideration is tested with respect to each of the examples. These example tests can be done in parallel, as was done in our previous work [8]. The level of exploitable parallelism is determined by the number of examples. The achievable speedup is less than the number of examples, however, as the example tests require different lengths of execution time. When the number of parallel example tests increases, the execution time tends to be dominated by the most time-consuming test.

Exploiting this type of parallelism requires a number of processors to run in parallel. It is therefore costly to exploit in terms of resources, but the benefit can be great. Memory contention is an issue when there is a large number of processors, as there might be more processors than memory banks. An efficient memory hierarchy is therefore needed in order to get the full benefit of all the processors. Individual pieces of the background data, upon which the individual processors operate, often refer to only one example. When this is the case each processor will only refer to a small well-defined subset of the data, something which can be exploited by the memory hierarchy.

Parallelism can also be exploited at a finer level, in the unification. During a unification two literals and their arguments are matched. In parallel unification, several arguments are unified in parallel. The speedup attainable in this way is bounded by the arity of the predicate being unified: for instance two 5-ary predicates can be unified at most five times as fast by using parallel unification.

The simple data structures we consider lead to simpler unification than in a more general Prolog setting, where the arguments can be nested structures. When all the data in the background knowledge are in the form of simple ground (non-variable) terms, unification can be achieved in a single step, involving either a register assignment or a comparison.

In order to take advantage of unification parallelism, several arguments must be fetched in parallel, thus requiring a higher memory bandwidth than sequential unification. Assuming n arguments are unified in parallel, the bandwidth can increase by a factor smaller than n. For sequential unification, the memory bus must be wide enough to accommodate the widest argument. With n arguments fetched in parallel, the bus must be wide enough to contain all the arguments, but some of these may be smaller than the widest argument. The increase in resource usage for unification hardware is proportional to the number of arguments fetched simultaneously, as each argument needs its own set of resources.

If only one word can be fetched from memory in a single cycle, parallel unification requires either memory banks that can be accessed in parallel, or that the arguments are packed into memory words.

4 Hypothesis Evaluation Hardware

Our proposed architecture is based on hypothesis evaluators. These are specialised unification processors unifying the literals in the hypothesis under consideration with the appropriate section of the background knowledge. The processors consist of hypothesis data registers, a background memory, a unifier unit, and an variable register file. These components are customised for the particular hypothesis space under consideration. The architecture of a single processor is shown in Figure 2.

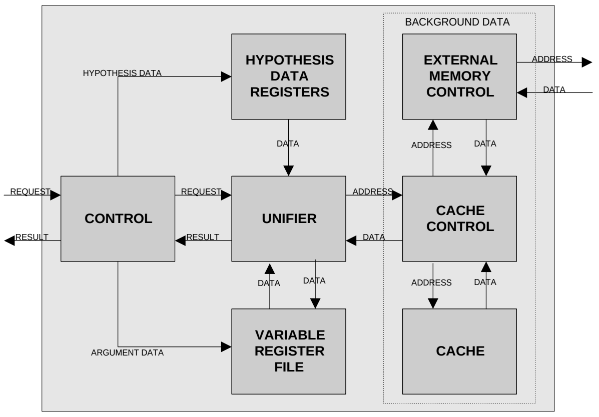

Figure 2 Architecture of a hypothesis evaluation processor.

The hypothesis data registers specify the types of literals occurring in the current hypothesis. For each argument of these literals, there is a register containing its type and a register containing its data (constant or variable number). The number of hypothesis data registers is limited by the number of literals that can occur in the hypothesis, as well as the maximal arity of literals in the hypothesis. The width of the registers is dictated by the data in the background knowledge.

The background memory gives access to the background knowledge. This memory is organised in a two-level hierarchy, with a small on-chip cache and a large shared off-chip memory. Issues regarding the memory architecture is discussed further in Section 5.

The unifier unit operates on data from the hypothesis data registers, background memory and from the variable register file. The arguments of literals found in the hypothesis data registers can be of four different types: output variable, input variable, void variable, and constant. The operation of the unifier is determined by the type of the argument in the hypothesis, found at compile-time, and can take one of the following forms:

1. Output variables are variables which are unbound at the time they are used. An output variable is bound by writing the argument from background memory to the register associated with the variable. This unification always succeeds.

2. Input variables are variable which are bound at the time they are used. Their value can be found in the appropriate register in the variable register file. Unification succeeds if the argument from the register file

is equal to the argument from background memory.

3. Void variables are variables occurring only once in a clause. The value of a void variable will not be needed again and since the it is unbound it matches anything. No action needs to be taken and unification succeeds.

4. For constants found in the hypothesis, unification succeeds if the constant is equal to the argument from background memory.

As described above, the unification of the arguments can be done in sequence or in parallel. The unifier matches data, or writes data to a register. In the sequential case this is done one after the other. In the parallel case this is done in parallel for a number of arguments, which requires parallel access to both hypothesis memory and the variable register file.

Following Prolog semantics, the processor performs a failure-driven loop over the background knowledge. The hypothesis memory contains data for each literal in the body of the hypothesis. The processor first attempts a match for the first literal in the hypothesis. If this search fails then the hypothesis does not explain the example, and the computation ends. If matching data are found, the processor moves on to the next literal in the hypothesis and pushes the next address to be read from data memory on a small stack. The variable bindings from the first literal are found in the variable register file. If the second literal succeeds, the processor continues with the third literal in the same manner, and so on until there are no more literals in the hypothesis. If a literal fails, when no matching data can be found, execution backtracks to the previous literal. Data are read from the address that is popped from the stack and are attempted unified with the previous literal.

As an example, consider the call of the first instance of *bond/4* from rule 3 in Figure 1 . The types of the four arguments are respectively: input variable, output variable, void variable, and constant. When the literal is called, the first variable is already instantiated and its value is found in the first variable register. Data memory now passes data from the appropriate section of background knowledge, such as *bond(110, 20, 28, 2)*. The unifier compares the first variable register with the first argument from background memory. It then writes the value 20 to the second variable register. It ignores the third argument. Finally it compares the fourth argument with the constant found in the hypothesis data registers. If the two comparisons are successful, the call succeeds and the unifier moves on to the next *bond* literal. At this time the bound value of the output variable is found in the variable register file.

In order to exploit query-level parallelism, we propose an architecture (Figure 3) containing a number of processors operating in parallel. Each processor has its own cache, but it may have to share external memory, depending on the number of processors and the number of external memory blocks. The architecture has a single main controller which interfaces to the main Progol host system. The controller receives hypothesis data from the Progol system on the host machine and passes it on to the individual processors. The processors read background data from the external memory one word at a time. Where RAM blocks are shared by more than one processor, access is controlled using a semaphore in the RAM control unit.

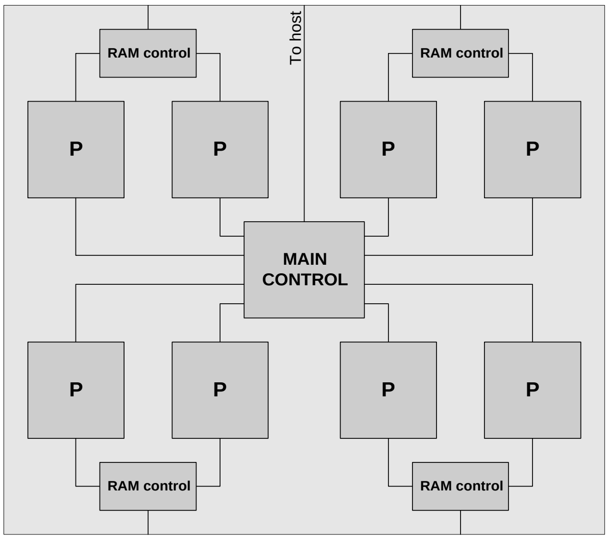

Figure 3 Architecture of a multi-processor system containing eight processors (labelled P) and four external memory banks, accessed by RAM controllers shared by two processors. The main control block links the Progol system to the individual processors. To avoid the main control unit becoming a bottleneck, it can be arranged to be a tree of controllers.

5 Caching

Adding a cache to the hypothesis evaluation processor has two effects. Firstly it shortens the execution time, as access time to external memory is longer than that to on-chip memory. Secondly it enables more processors to run in parallel, as the number of accesses to external memory drops.

The effectiveness of the cache depends on the amount of temporal and spatial locality in the background data. A hypothesis test consists of calling the hypothesis body literals in turn. Body literals referring to different predicates also refer to different parts of the data. The entry point into the data for a particular literal is determined by its index, typically its first argument. The call to the literal iterates over all the data with the same index. Spatial locality is ensured by grouping the data according to the index. A literal is likely to be called several times. If a call fails, execution backtracks; if an alternative solution is found for a previous body literal, the failed literal is called again. Temporal locality is available where the literal is called again with the same index.

As an illustration of the behaviour of the cache, consider rule 2 in Figure 1. This has two calls to *atm/5*, followed by a call to *bond/4*. All the body literals use the same index. Data are first fetched from the section of the *atm*-data referred to by the index A. Data are fetched until a matching atom is found. At this point the cache is partially filled up with *atm*-data. When a match is found, *atm* is called again with different arguments. Since this call uses the same index and therefore the same data, the initial fetches can be made from the cache rather than from the external memory. If the second *atm*-call finds a match, *bond* is called. The literal *bond* refers to different data than the two previous calls, and the behaviour of the memory system depends on whether there is a separate cache for *bond* or not. In the first case the *bond*-cache will fill up. If there is a failure, one of the *atm*-calls is likely to provide an alternative solution. When *bond* is re-entered it can still use the old contents of the cache, since the index has not changed. If there is one cache, *bond* will overwrite the value of the *atm*-data. Upon backtracking, *atm* is likely to miss the cache, and re-entering *bond* may also cause some misses. Since the call to *bond* forms the inner loop, the cache will end up with the constantly re-used *bond*-data.

The size of the cache can be predicted from the nature of the background knowledge. Assuming most time is spent on the inner loop where the same data are repeatedly reused, the cache should be large enough to contain all data referred to by the same index. For the mutagenesis data set, both the *bond/4* and *atm/5* predicates are defined by some 6000 clauses each, but no more than 44 entries are referred to by a single index. The data entries themselves require 23 and 58 bits respectively, with a 32-bit floating point number in *atm/5*.

The block placement strategy also needs to be addressed. A directly mapped cache can be used when all the background data are accessed by a processor in consecutive memory words, or when cache size is dictated by the maximum number of entries accessed by a processor. This has the advantage of reduced hardware usage compared with associative caches. A direct mapped cache will result in some conflict misses as calls to different predicates overwrite data which may be needed later, as in the example above. Associating each predicate which can be called in a hypothesis with a separate cache reduces the conflict misses; indeed it can eliminate them entirely, if each call to the same predicate uses the same index.

Cache architectures typically have multiple levels. We consider only one level of off-chip RAM in addition to the embedded RAMs. These memories can be organised into a two-level hierarchy, with level one on-chip and level two off-chip. Alternatively, several levels of cache can be built from the on-chip RAMs. With each processor having its own cache, adding a second-level cache reduces the number of processors which can be put on the chip. The speedup gained by adding the second level must therefore offset the effect of this lost parallelism. In the two-level cache, the first level contains the entire active set for the inner loop. A second level between this and the external memory can reduce the number of conflict misses occurring in a hypothesis like the example above, where a call to *atm* after failure in *bond* will find its own data overwritten. In order to guarantee that the *atm*-data are found in the second-level cache at the time of backtracking, the second-level cache needs to be twice as large as the first level. This increases memory usage by a factor of three, but does not reduce access time to the first level. The speedup gained by reducing conflict misses will not be significant, as most time is spent on the inner loop which has all its data in the first-level cache after an initial streak of cold-start misses.

The accesses to the external memory can be either demand-driven or based on prefetching. The demand-driven approach reduces the number of accesses to external memory, as only what is needed is fetched. Since the data accesses are predominantly sequential, whole sections of data can be prefetched. This is especially attractive for external RAM optimised for burst access. With the dual-ported embedded RAMs found on some FPGA chips, the cache controller can write data from external memory while the processor uses the existing data.

6 Compilation Flow

There are three stages in the compilation flow.

Data preprocessing. In order to optimise the data and gather information to guide hardware compilation, we perform the following steps:

1. Group data according to their predicate.
2. For each predicate, group data according to the first argument.
3. Determine the maximum required size of the cache, by finding the largest number of clauses with the same first argument.
4. Map constants to numerical values and do bit-width analysis.

5. Define a packing scheme for each predicate based on the bit-widths of its arguments.
6. Generate an indexing table for each predicate, mapping all the possible first arguments to the offset into the data for that predicate.

We currently have tools to estimate the cache size, do bit-width analysis, data packing, and produce the index table.

To illustrate data preprocessing, consider the background knowledge in the mutagenesis data set consisting of 12 000 facts defining the predicates *atm* and *bond*. After sorting the background knowledge the number of occurrences of every predicate is found, and the maximum (44) dictates the depth of the cache. Constant values, e.g. *c* and *h* are mapped to numerical values. After this mapping, we determine the maximum number of bits required for each argument. For the *bond* predicate, this is 8, 6, 6, and 3 bits respectively. Given the depth of the cache and the width of the embedded RAMs, we find that all arguments can fit in a single word. When the packed data are output, each first occurrence of the indexing argument is recorded in the index table.

Compilation. Based on information from the preprocessing stage and user supplied information, the processors are generated. Progol requires the user to specify the elements from which hypothesis can be generated, and the maximum length of hypothesised clauses. From these we find the parameters for the control of the unifier unit and the number of registers in the variable register file and hypothesis data memory. Although it is hard to estimate the size of the placed and routed design before compilation, we can determine an upper limit imposed by the caches. The total number of processors must be small enough for the combined size of the caches not to exceed the total available embedded memory.

Run time. At run time, Progol generates hypothesis to test. Each hypothesis is encoded and passed to the main controller which in turn passes it on to the individual processors.

7 Results

We have tested different versions of the proposed architecture. Our evaluation implementations have a variable number of processors. The architectures are designed using the Handel-C hardware description language, and are simulated using the DK Handel-C simulator. From this the cycle counts for our benchmarks are found.

The cache has a latency of two cycles and cache access is not pipelined. The external memory is estimated to have a latency of six cycles, including access arbitration for shared off-chip RAMs. Accesses to external memory from different processors are pipelined, so the maximum throughput is one data word per cycle.

We use the mutagenesis data set for benchmarking. In testing the effect of caching, we use the nine final rules generated by Progol (1692 queries), while the other results are based on rule 3 from Figure 1 only (188 queries). Six of the nine rules contain a single body literal, one rule contains two body literals, while the last two rules (including rule 3) contain three body literals.

We target the RC1000 platform with a Xilinx XCV2000E chip. Tools from Xilinx estimate the clock speed and area usage. For benchmarking we use an architecture without hypothesis data memory clocked at 38MHz. A single-processor architecture with hypothesis data memory and sequential unification can be clocked at 34MHz, and requires 1000 slices out of a total of 19200. The XCV2000E chip can accommodate 24 processors, using 46 out of 160 available 4Kb BlockRAMs.

7.1 Single processor caching

The effect of adding a cache depends on the rules in the nine-rule benchmark. The shorter rules benefit little, since there is less temporal locality. In an uncached single-processor architecture the external memory utilisation lies between 0.16 and 0.19. With a five cycle penalty on cache misses, the utilisation of the external RAM decreases by a factor of ten for rule 3. For rules 2 and 4 the utilisation decreases by a factor of two, while the rest are one-literal rules which show no improvement.

The cost of misses also affects the speedup that can be attained by using a cache. Figure 4 shows the speedup for the nine rules where the cache access cost is one cycle and the cache miss cost is up to 30 cycles. The thick line shows the combined effect for the nine rules, while the top line shows the effect for rule 3.

The above discussion indicates that the overall performance of the system depends on the type of generated hypothesis. Short rules result in low cache performance which in turn limits the amount of query-level parallelism which can be extracted, as memory access becomes a bottleneck. The rules we use, however, are the final rules produced by Progol. During the hypothesis search we expect that longer rules will dominate, because such rules are abundant in the hypothesis space. Further testing on run-time data is needed to confirm this expectation.

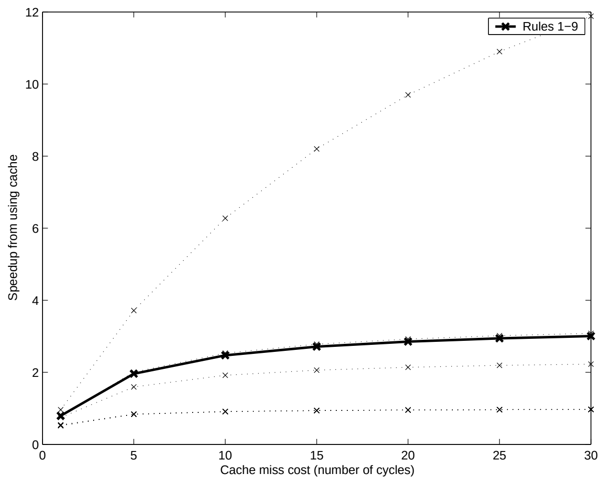

Figure 4 Speedup of cached architecture over an uncached version. The x-axis shows the cost of cache misses in cycles, while the y-axis shows the speedup. The dotted lines display results for individual rules in our benchmark, while the thick solid line shows the aggregate result.

The limited size of our benchmark has to be noted. It covers a single hypothesis, but we expect similar results to hold for a larger set of hypothesis. The full hypothesis space consists of rules of different lengths. Some of these rules contain only a single literal in the body. For these rules caching will have little effect, as each piece of data is only used once. At the same time the shorter rules take shorter time, as there is only one pass over the data. Caching will have stronger effect on rules where there are repeated calls to the same predicate, as most of the memory accesses from the second instance will be cache hits.

7.2 Query parallelism

We carry out hardware simulations for architectures containing up to 64 processors. All processors share a single external memory bank. The results are shown in Figure 5 with the speedup relative to a single-processor architecture. This shows a good speedup initially, which then drops off somewhat. At 32 processors the speedup is 17 while at 64 processors the speedup is 19. This simulation provides similar results to our previous work, which shows diminishing returns of adding processors since long computations dominate. All processors are dedicated to a single hypothesis at a time, and the queries vary in the length of time they run for. Therefore, as the number of processors approaches the number of examples to be tested, some processors will be idle while the more demanding example tests are carried out on other processors.

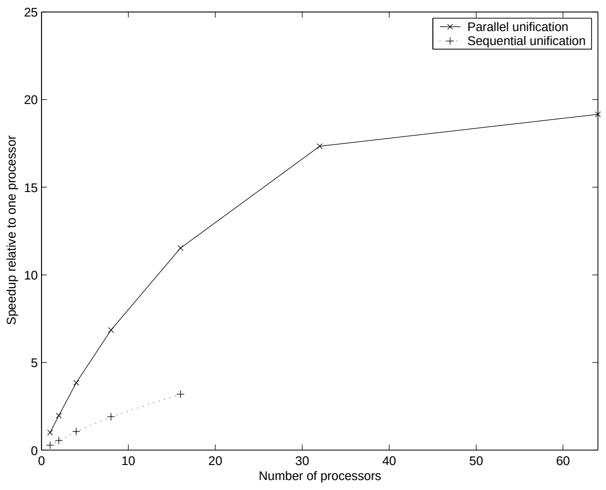

Figure 5 The effect of coarse-level parallelism on speedup. The x-axis shows the number of processors, while the y-axis shows the speedup relative to a 1-processor architecture with parallel unification. Both parallel and sequential unification architectures are shown.

7.3 Unification parallelism

We have benchmarked architectures containing both sequential and parallel unification processors. Parallel unification is on average 3.6 times faster in terms of the number of cycles. Each unification is speeded up by a factor at most equal to its arity. In our benchmark, the bond predicate dominates the run time. This predicate has an arity of four.

Note that for this predicate all the data for a single clause could fit into one memory word, so unification could be done in one cycle rather than four. For other programs the data may not be so compact. Either wider data or a higher arity can result in the clause requiring several memory words, so the effect of parallel unification is smaller.

7.4 Comparison with software

Our architecture compares favourably with software running on a microprocessors. A Pentium 4 1.8GHz computer running our benchmark, using Progol's built-in Prolog evaluator, completes our benchmark in 0.095 seconds. At 38MHz a one-processor implementation with parallel unification completes the benchmark in 0.0102 seconds, 9 times faster. A 32-processor implementation can complete it 160 times faster than the Pentium, if it runs at 38MHz. This benchmark is a subset of a full run of the mutagenesis problem. The exact run-time for this depends on the search-space parameters, and is measured in hours.

7.5 Comparison with instruction processor

The architecture presented in this paper is based on a direct datapath implementation. In a previous implementation [8], we adopt an instruction processor approach to speed up Progol. A single instruction processor takes 13 million cycles to complete our benchmark. Our current architecture takes 390 000 cycles (33 times fewer) using parallel unification and 1.4 million cycles (9 times fewer) using sequential unification. The clock speeds are similar, 35MHz and 38MHz respectively. Our current size estimates indicate that a single processor with sequential unification requires around a quarter of the space of the instruction processor.

The datapath architecture benefits from using a simpler unification. This results in lower hardware usage, which can be exploited by having more parallel processors. Also, the datapath architecture does not need the control structures of the instruction processor. These result in a higher memory usage. An advantage of the instruction processor is that it can handle a wider range of data, as it is not constrained to a hypothesis space referring only to ground unit clauses in the background knowledge.

8 Conclusions and Future Work

We have demonstrated an architecture capable of executing inductive logic programs at high speed. The architecture performs well both when compared with inductive logic programming software executing on a modern microprocessor, and with an earlier architecture [8] based on multiple instruction processors targeting the same problem domain. To recapitulate the main points:

- The architecture is based on a direct mapping of the datapath. This has a great speed advantage over instruction processor systems, and is also shown to have good resource utilisation.
- Parallelism is exploited at several levels. This leads to a multi-processor architecture with good speedups. Preprocessing the input data enables us to exploit unification parallelism in all the processors, for a uniform speedup.
- By using a cache system, we reduce access time to memory, and more importantly we reduce memory contention sharply. This allows more processors to run effectively in parallel. The sizes of the caches are found by analysis of the input data.

Current and future work consists of the following. First, improve the interface between the main controller and the processors – the current interface can result in long connections and a large fanout when many processors are used. This problem can be overcome by pipelining the connections and decomposing the main controller into a tree network of controllers. Second, explore strategies for parallelising communication between the cache and the external memory. Third, evaluate our architecture using a wide range of Progol applications.

Acknowledgements. The support of Celoxica Limited, Xilinx, Inc., the UK ORS Award Scheme, and the UK Engineering and Physical Sciences Research Council (Grant number GR/N 66599, GR/R 31409 and GR/R 55931) is gratefully acknowledged.

References

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

[2] S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.

[3] M. Turcotte, S. Muggleton and M. Sternberg. Protein fold recognition. *Proc. 8th International Workshop on Inductive Logic Programming (ILP-98),* LNAI 1446, Springer-Verlag, pp. 53–64, 1998.

[4] 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, GMD-Studien Nr 237, 1994.

[5] P. Finn, S. Muggleton, D. Page and A. Srinivasan. Pharmacophore discovery using the inductive logic programming system Progol. *Machine Learning*, 30:241-271, 1998.

[6] H. Ohwada, H. Nishiyamai and F. Mizoguchi. Concurrent execution of optimal hypothesis search for inverse entailment. *Proceedings of the Tenth International Conference on Inductive Logic Programming,* pp. 165–173, 2000.

[7] D. Skillicorn and Y. Wang. Parallel and sequential algorithms for data mining using inductive logic. *Knowledge and Information Systems*, 3:405-421, 2001.

[8] A. Fidjeland, W. Luk and S. Muggleton. Scalable acceleration of inductive logic programs. *Proc. Int. Conf. on Field-Programmable Technology (FPT'02),* IEEE, 2002.