Customising Application-Specific Multiprocessor Systems: a Case Study

Andreas Fidjeland and Wayne Luk
Imperial College London
180 Queen's Gate
London SW7 2BZ
{akf,wl}@doc.ic.ac.uk

Abstract

This paper reviews the development of application-specific multiprocessor systems for machine learning applications, and indicates how variants of such systems can be produced by design customisation. We first provide an overview of Progol, a machine learning framework based on Inductive Logic Programming. We then describe, for such frameworks, various uniprocessor architectures and their adoption in multiprocessor systems. We also present the experimental facilities and results for evaluating our approach, and a method for automating the compilation of such designs.

1 Introduction

The ASAP Conference began in 1986 as the International Workshop on Systolic Arrays. The current focus on application-specific systems reflects a broadened theme to take into account recent technological advances. One of the most important of such advances is field-programmable hardware technology: its increasing sophistication enables novel application-specific systems to be developed rapidly at affordable cost.

A field-programmable hardware device delivers a typical abstraction to its users as an array of programmable logic and storage elements, linked together by programmable connections. In the early days when field-programmable hardware had limited capacity, systolic architectures provided an effective way of exploiting the available resources. More recently, the rapid improvement in capacity and capability of field-programmable hardware has brought exciting opportunities for hardware support of innovative architectures, in addition to systolic ones.

One promising direction is to deploy field-programmable hardware in application-specific multiprocessor systems. The use of such systems for logic emulation [2] is well-known; other applications include video processing [12], network processing [15], and scientific computations [17].

In this paper, we review the development of application-specific multiprocessor systems for machine learning applications, and indicate how variants of such systems can be produced by design customisation. Machine learning has a wide range of applications, including satellite fault diagnosis [5], finite element mesh analysis [3], and protein folding in molecular biology [16]. While machine learning based on a high-level framework, such as the Progol Inductive Logic Programming approach, has shown good promise, there are significant performance bottlenecks which we hope our multiprocessor system would be able to overcome.

The rest of the paper is organised as follows. We first provide an overview of Progol, a machine learning framework based on Inductive Logic Programming (Section 2). We then describe, for such frameworks, various uniprocessor architectures (Section 3) and their adoption in multiprocessor systems (Section 4). We also present the experimental facilities (Section 5) and results (Section 6) for evaluating our designs, and a method for automating the compilation of such designs (Section 7). We shall end with concluding remarks and suggestions for further work (Section 8).

2 Background: Progol

Inductive logic programming (ILP) [10] is a learning paradigm based on first-order logic. ILP systems produce predicate hypotheses from background knowledge and examples. One advantage of this approach is that both the input background knowledge and the output hypotheses are in a human-readable format. Another advantage is that by incorporating background knowledge, the learning system can build on partial theories.

In general an ILP algorithm takes as input a set of pos-

itive and negative examples (E), some background knowledge (B) and a language bias which defines the hypothesis space. From this it produces a theory (H) explaining the examples:

$$B \wedge H \models E$$

ILP algorithms search through a hypotheses space, iteratively assessing the quality of the current theory and specialising or generalising as appropriate. ILP algorithms employ different strategies for constructing and searching through the hypothesis space and in assessing the quality of each hypothesis.

The ILP system Progol [9] is based on a method called mode-directed inverse entailment. Progol constructs a lattice of potential hypothesis bounded above and below by a most and a least general hypothesis. The search space is explored using an A*-like algorithm. The searching and testing is computationally demanding, and learning tasks can run for hours or days on modern workstations. To avoid overfitting which produces excessively specific solutions, cross-fold validation is used which increases execution time further.

In order to cope with the computational demands, different approaches to parallelising the learning process have been proposed, such as splitting up the data set and form hypotheses based on each partition [13], partitioning the language bias [4], and performing parallel branch and bound search through the hypothesis space [11]. There is also scope for parallelising the quality assessment of hypotheses, a method that we exploit in our architectures.

Progol is an extension of the logic programming language Prolog. We give only a brief and informal introduction to the required terminology. Prolog is a declarative language where a program consists of a collection of facts and rules. A *predicate* defines a relation, and is defined by one or more *clauses*. Each clause contains one or more *literals*, generally organised as a *head* literal and zero or more *body* literals. A literal consists of a functor and arguments. The arguments can be constants or variables. A *unit* clause contains only one literal. A *ground* clause contains no variables.

Prolog execution is characterised by the use of unification to pass arguments, and the use of non-deterministic execution. A computation can fail, at which point it *backtracks* to an earlier state to try an alternative solution. A procedure call can have many alternatives, and clause indexing is commonly used to reduce the number of possible calls.

3 Uniprocessor systems

We have developed several processors for use in the Progol learning system. We speed up the hypothesis testing phase of Progol, which tests whether or not a hypothesis instance is explained by the background knowledge. Both the hypothesis and the background knowledge are *Prolog* programs. The processors are therefore Prolog processors executing a stream of queries (the hypothesis) with a program (the background knowledge) loaded into memory.

3.1 WAM instruction processor

Our first processor design [7] is based on the Warren Abstract Machine (WAM) execution model for Prolog [1],[18]. This is an execution model which is widely used in software implementations of Prolog. The abstract machine is a stack-based machine based on a high-level instruction set which maps closely to Prolog code.

Our instruction processor differs somewhat from the original WAM. The main differences are that the control and data instructions are kept separate, all data-oriented instructions are replaced by a monolithic unify operation, and bindings are stored on a separate stack. As in the original WAM a local stack keeps both determinate and non-determinate activation records, the trail stack records bindings which need to be undone upon backtracking, and a small unification stack holds temporary data during unification of complex terms. The queries generated by the host are stored in a reserved section of the code and data segments in the main memory.

The execution of the complex WAM instructions is not pipelined, but the control logic for each instruction is highly parallelised and makes use of speculation. The current implementation does not cache instructions or data.

3.2 Data processor

Our second processor design [6] consists of a data processor architecture which exploits simplifications in the background knowledge. Specifically, it is assumed that the background knowledge contains ground unit clauses only. The lack of variables in the background knowledge means that unification can take one of only a few forms, and a simpler architecture can therefore be used.

The basic operation of a data processor is the same as before: receive a hypothesis and evaluate it with respect to the background knowledge. The processor contains a small hypothesis memory, background memory cache, a unifier unit, and a variable register file. These components are customised for a particular hypothesis space. The architecture of this processor is shown in Figure 1.

The unifier unit operates on tagged data from the hypothesis memory, background memory and the variable register file. The four data types in the hypothesis data correspond to four unification operations (register store, register compare, input data compare, no-op). The literal and argument fields

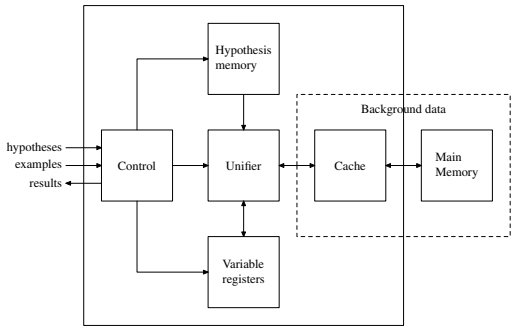

Figure 1. Architecture of data processor

of the hypothesis contain relevant control information, and are used in a simple controller containing a few counters and a minimal stack.

We observe that data access to segments of the background knowledge with the same index display a high degree of temporal locality. We make use of this observation to make an estimate of the cache resource requirement in the data set. For a data set, we examine the number of clauses in the background knowledge using the same index, and use the maximum value as the size of the active set. The input data are grouped by index in order to improve cache performance.

3.3 Customisable instruction processor

The two processor designs presented above each has its benefits: the instruction processor can handle any form of input data, while the data processor is much smaller and faster. Our third processor design, Arvand, is developed to combine the benefits of both by using a customisable processor design. It is an instruction processor using a different execution model which allows a simplified version of the processor to resemble the data processor, while a more complex processor, using the same instruction set, can handle the full range of input data.

We make use of the Vienna Abstract Machine (VAM) [8] execution model for Prolog. The VAM uses two instruction streams in order to interleave argument setup and unification. This execution model ties in well with the execution model used in the data processor, where two streams (of tagged data) are unified.

The Arvand processor operates on two instruction streams in a four stage pipeline. The instruction streams, one for the head and one for the goal, are cached independently in direct-mapped caches. The deterministic and non-deterministic control stacks are kept on the local stack in the main memory, while the top part of the stack is also buffered locally. The register file contains the variables for the head, while the topmost frame in the stack buffer con-

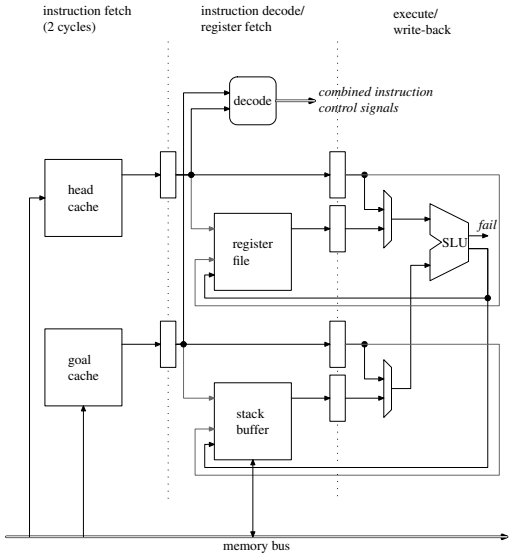

Figure 2. Customisable Arvand processor architecture overview

tains variables for the goal. The topmost choice point on the non-deterministic stack is also kept in registers; this allows fast activation record push and pop operations, which is important for shallow backtracking. If complex structures are used, the unifier (labelled SLU in Figure 2) builds temporary data on the global stack. Additional execution units are required for extra-logical predicates. Currently the only such execution unit is an accumulator-based ALU. Figure 2 shows an overview of the datapath of the processor, without the optional ALU, or the global stack buffer.

The use of programmable logic as the implementation platform makes it possible to cheaply and quickly generate architectures. Variations of the basic architecture can therefore be used to cater for the specific needs of different input programs or data. These customisations can be made in the dataflow of the processor, or by parameterisation of memory units. The caches and buffers in the processor can all be customised to the input program. Some interfaces between the internal processor storage elements and the main memory are optional. Some classes of programs require local buffers only, for goal instructions and the local stack; while others may not require the global stack. The decode unit can be tailored to cater only for the required instruction combinations. For programs which deal with symbolic data only, the ALU can be removed.

4 Multiprocessor architectures

To perform tasks in parallel we combine processors into a multiprocessor. The multiprocessor is in the form of a pro-

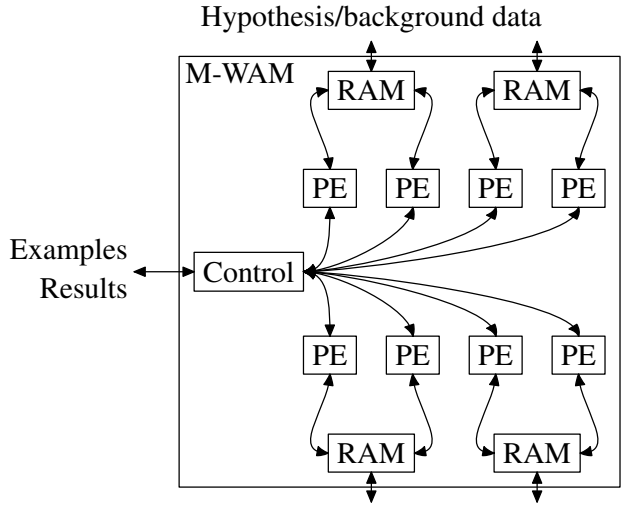

Figure 3. Multiprocessor architecture for Progol

cessor farm, where a controller node receives or generates tasks and dispatches them to the available processors.

The main memory, of which there may be several banks, is shared between processors. An individual processor may have its own internal caches and buffers and may also have private data structures stored in the shared external memory. The same code segment is shared by all the processors and is replicated across the available memory banks.

The most basic multiprocessor architecture contains identical processors. Even with this homogeneous architecture, different configurations are possible, with the important trade-off being between the number of processors and the cache size. At one end of the range of possibilities the multiprocessors contain a large number of parallel processors with small caches, while at the other end they contain a small number of processors with large caches.

Two effects interact to determine the speedup that can be attained for a given number of processors. First, increasing the number of processors increases the level of parallelism for the independent parts of execution. Some part of the parallel tasks needs to be sequentialised, and this imposes an absolute limit on the speedup that can be attained. Since the processor nodes in the multiprocessor share the main memory bus, the bandwidth of this bus is a limiting factor. The speedup gained from the parallel execution of the independent parts of the tasks will tend to converge on an upper limit determined by the available memory bandwidth and the aggregate memory bandwidth requirements for all the processors.

Second, increasing the number of processors decreases the performance of each processor. With a fixed amount of resources, increasing parallelism results in less resources per processor. In our multiprocessor, the cache size is dependent on the number of processors. More processors implies smaller caches which in turn implies higher memory bandwidth requirements. Increasing the number of processors will therefore tend to increase the proportion of computation which must be sequentialised, resulting in lower performance.

5 Experimental setup

We use two different target platforms, RC1000 and RC2000. The WAM instruction processor and the data processor are both implemented on the RC1000 platform. This contains a Xilinx XCV2000E FPGA chip, with 160 embedded 256 $\times$ 16 bit RAMs. The board contains four external RAM banks. The Arvand processor targets the RC2000 platform. The RC2000 contains a Xilinx XC2V6000 FPGA chip, which contains 144 embedded 512 $\times$ 36 bit RAMs, used for caches and buffers. The board also has 6 external RAM banks.

For the Arvand processor, we have performed cycle-accurate execution of single-processor runs. Execution traces for cycle time and memory usage is used as a basis for simulating multiprocessor performance.

For comparisons with software, we compile the benchmarks to native code for x86 using GNU Prolog. The software runs on a 2GHz Pentium 4 processor, with 256MB RAM and a 512k L2 cache.

We construct benchmarks based on two data sets from bio-informatics: protein folding and mutagenesis. The protein folding data set is taken from [16], where Progol is used to predict the tertiary structure of proteins. We use the data for the immunoglobulin fold. The mutagenesis data set [14] is used in a study to predict mutagenic activity, linked to carcinogenity, of nitro-aromatic compounds. Both benchmarks are constructed by sampling the hypothesis generated by a full run of Progol.

For the Arvand processors we use two configurations. The first deals with complex data, while the second one deals only with ground unit clauses in the background knowledge, like the data processor (Section 3.2). The complex processors are named Cn where n is the number of bits required to address the head cache. Similarly, the simple processors are named Sn. The C-processors are used for the immunoglobulin data set, while the S-processors are used for the mutagenesis data set.

6 Results

We compare the execution time of the Arvand uniprocessor and multiprocessor with software, as described in the

previous section. For the uniprocessors the performance is roughly half that of the Pentium 4 processor. For multiprocessors we observe a ten times speedup using 38 processors.

The WAM processor is the largest design. Four of these processors can fit on the XCV2000E chip. For the data processor, 32 processors can fit on a single XCV2000E chip. The Arvand processors come in several configurations. The complex configuration allows 16 processors on the XC2V6000 chip, with embedded RAMs being the limiting factor. The simple configuration allows 38 processors on the same chip, with configurable fabric being the limiting factor. The XC2V6000 chip is around three times larger than the XCV2000E.

In terms of execution time, the Arvand processor and the data processor are the fastest. The data processor is observed to be around 30 times faster than the WAM instruction processor on the same data set. The simple configuration of Arvand has similar execution time as the data processor.

The speedup that can be attained for different multiprocessor configurations varies between data sets. Figure 4 shows the speedup for the different Arvand processor configurations for the immunoglobulin and mutagenesis data sets. The speedup is calculated relative to a single 4k cache processor. In addition, the aggregate miss rate is plotted, calculated based on the total miss rate of the single processor run.

The speedup correlates with the number of processors, but the effect of higher miss rate for small cache sizes limits the speedup. For the immunoglobulin data set, the critical point is in the switch from 4k to 2k cache. For the mutagenesis data set, the active set can fit in even the small 1k caches, and performance improves steadily, but the speedup is limited by the longest running task.

From the above results, the link is clear between the aggregate miss rate of the multiprocessor and its overall performance. For one data set the performance limitation is imposed by the adverse effect of high aggregate miss rate. In this case a good estimate of the cache requirements is imperative to attain high performance and good resource utilisation.

7 Design automation

The architectures presented above are hand-crafted. In order to speed up development of multiprocessor architectures for inductive logic programming and related domains, we are now developing an architecture generation framework. This framework makes use of a high-level logic programming oriented architecture description language, Archlog. The language and its compilation framework has the following benefits:

- *Ease of development.* A high level of abstraction allows fast development of architectures and encourages architectural exploration. The use of a language close to that already in use in the intended application domain encourages its use by software application developers.

- *Separation of concerns.* The architecture generation is library based, enabling application-developers to work at a high level of abstraction, while the hardware developers focus on the efficient realisation of the library components.

- *Flexible hardware/software partitioning.* The hardware description and the description of the software running on the architecture are similar, facilitating easy migration from software to hardware, either manually or automatically.

The Archlog language is similar to Prolog, and is used to hierarchically describe an architecture. A special *fork* construct is used to express data-parallel operation. A number of processing primitives provide the basic building blocks of an Archlog architecture. These can be quite simple, like an adder operating on a stream. However, the main processing element remains the customisable Arvand processor from Section 3.3.

The synthesis in the Archlog framework is library-based. The Archlog description is transformed to a netlist containing processing elements and communication units with standardised interfaces. The processing elements occur in the Archlog program, but the communication units, such as pipeline buffers and memory interfaces, must be mostly inferred. Program analysis of the input software is used to set parameters for the hardware generation.

We expect to be able to use the Archlog framework to produce optimised designs quickly. The framework will then be used to implement different inductive logic programming architectures, as well as architectures for other logic programming based applications.

8 Conclusions

We have described the development of application-specific multiprocessor systems for machine learning applications, particularly for the Progol framework based on Inductive Logic Programming. We demonstrate that although our customised multiprocessor system runs at a lower clock speed, it can out-perform traditional microprocessors by up to an order of magnitude. Current and future work includes refining our multiprocessor architecture for improved effectiveness, and investigating how run-time reconfiguration can be adapted for design optimisation.

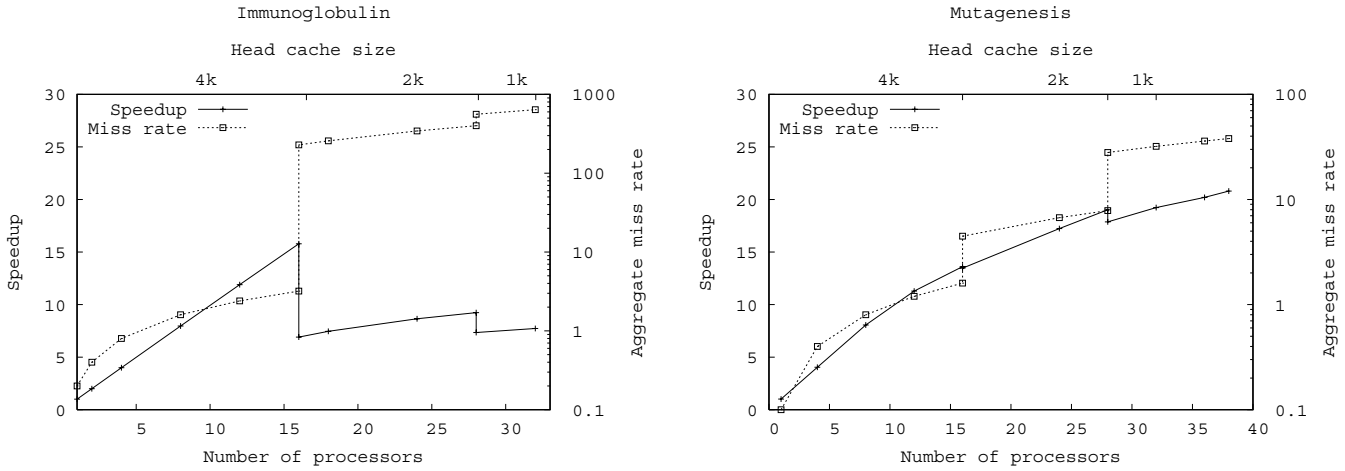

Figure 4. Performance of homogeneous multiprocessors

References

[1] H. Aït-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction*. MIT Press, Cambridge, MA, USA, 1991.

[2] L. A. Barroso, S. Iman, J. Jeong, K. Oner, K. Ramamurthy, and M. Dubois. RPM: A Rapid Prototyping Engine for Multiprocessor Systems. *IEEE Computer Magazine*, February 1995.

[3] I. Bratko and S. Muggleton. Applications of inductive logic programming. *Communications of the ACM*, 38(11):65–70, 1995.

[4] L. Dehaspe and L. D. Raedt. Parallel inductive logic programming. In *Proc. MLnet Familiarization Workshop on Statistics, Machine Learning and Knowledge Discovery in Databases*, pages 112–117, Heraklion, Crete, January 1995.

[5] C. Feng. Inducing temporal fault diagnostic rules from a qualtitative model. In *Proc. Int. Workshop on Machine Learning*, pages 403–406, San Mateo, C.A., 1991. Morgan Kaufmann.

[6] A. Fidjeland and W. Luk. Customising parallelism and caching for machine learning. In *Proc. IEEE Int. Conf. Field-Programmable Technology*, pages 204–211, Tokyo, Japan, December 2003.

[7] A. Fidjeland, W. Luk, and S. Muggleton. Scalable acceleration of inductive logic programs. In *Proc. IEEE Int. Conf. Field-Programmable Technology*, pages 252–259, Hong Kong, December 2002.

[8] A. Krall and U. Neumerkel. The Vienna Abstract Machine. In *Proc. Int. Workshop Programming Language Implementation and Logic Programming*, number 456 in LNCS, pages 121–135. Springer-Verlag, August 1990.

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

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

[11] H. Ohwada, H. Nishiyamai, and F. Mizoguchi. Concurrent execution of optimal hypothesis search for inverse entailment. In *Proc. Int. Conf. Inductive Logic Programming*, pages 165–173, 2000.

[12] P. Sedcole, P. Y. K. Cheung, G. Constantinides, and W. Luk. A reconfigurable platform for real-time embedded video image processing. In *Field Programmable Logic and Applications*, number 2778 in LNCS, pages 606–615. Springer-Verlag, 2003.

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

[14] A. Srinivasan, S. Muggleton, R. King, and M. Sternberg. Mutagenesis: ILP experiments in a non-determinate biological domain. In S. Wrobel, editor, *Proc. Int. Inductive Logic Programming Workshop*. Gesellschaft fur Mathematik und Datenverarbeitung MBH, 1994. GMD-Studien Nr 237.

[15] S. Subramanian and C. Gloster Jr. Mapping networking applications to multiprocessor-FPGA configurable computing systems. In *Proc. MAPLD*, 2003.

[16] M. Turcotte, S. Muggleton, and M. Sternberg. Protein fold recognition. In C. Page, editor, *Proc. of the 8th International Workshop on Inductive Logic Programming (ILP-98)*, LNAI 1446, pages 53–64, Berlin, Springer-Verlag, 1998.

[17] X. Wang and S. Ziavras. A configurable multiprocessor and dynamic load balancing for parallel LU factorization. In *Proc. Int. Parallel and Distri. Processing Symp.*, 2004.

[18] D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, USA, October 1983.