A Customisable Multiprocessor for Application-Optimised Inductive Logic Programming

Andreas Fidjeland, Wayne Luk and Stephen Muggleton
Imperial College London, 180 Queen's Gate, London SW7 2AZ, United Kingdom
akf@doc.ic.ac.uk,*wl@doc.ic.ac.uk*,*shm@doc.ic.ac.uk*

Abstract

This paper describes a customisable processor designed to accelerate execution of inductive logic programming, targeting advanced field-programmable gate array (FPGA) technology. The instruction set and the microarchitecture of the processor are optimised for key operations in logic programming, such as unification and backtracking. Such optimisations reduce external memory access to enable performance comparable to current general-purpose processors, even at much lower clock frequencies. Our processor can be customised to a particular program by excluding unnecessary functional and memory units, and by adapting the size of such units to suit the application. These customisations reduce resource usage while improving performance, and enable accommodating multiple processors on a single FPGA. Such multiprocessor parallelism can be exploited by search-oriented applications in machine learning applications. We find that up to 32 processors can fit on an XC2V6000 FPGA. Using this device, the computational kernel of the machine learning system Progol, when applied to common bioinformatics data sets for learning to identify mutagenesis and protein folds, can yield speedups of up to 15 times over software running on a 2.53GHz Pentium-4 machine. The proposed approach appears promising with the advance of field-programmable technology: the more recent XC4VLX160 device would be capable of supporting up to 65 processors.

Keywords: Application-specific processors, Processor customisation, Inductive Logic Programming

1. INTRODUCTION

Current general-purpose processors are powerful devices which are generic enough that they can be applied to many different application domains. Enormous effort has been dedicated to achieving high performance for single-core processors and, more recently, for multi-core processors. However, dedicated hardware can outperform the fastest general-purpose processors by being optimised for certain tasks. For example, functionality such as graphics and network processing has moved from the general-purpose processor to specialised co-processors. The tasks which have moved to specialised units have been justified by high performance demands and high volumes of production.

We aim to speed up inductive logic programming [7], a form of symbolic machine learning. Inductive logic programming has found many uses in bioinformatics, for example to find rules governing properties such as protein folding [15], mutagenic activity [12], and structure-activity relationships [13]. These applications are often computationally demanding, as inductive logic programming explores large search spaces where computations can run for hours or days.

While the inductive logic programming domain requires high performance, the size of the domain is fairly small. We therefore aim to accelerate the application domain using reconfigurable architectures. Such architectures can be programmed to implement different functionality, and have been successfully used to implement algorithms directly in hardware. Inductive logic

programming can be seen as an extension of logic programming, and we therefore base our solution around soft instruction processors, i.e. processors implemented in reconfigurable fabric.

Soft processors have disadvantages compared with general-purpose microprocessors, in particular clock speed and resource usage. However, our approach is promising since: (a) the underlying execution model is different from general-purpose processors, so a specialised processor can perform more computations per cycle; (b) FPGAs enable customising the processor architecture for a particular problem instance, hence reducing resource usage while improving speed; (c) inductive logic programming is easily parallelisable, so a chip multiprocessor, with processors on one chip, can exploit the increasing capacity and capability of FPGAs. Indeed, similar approaches have been adopted for exploring various aspects of computer architecture, such as transactional memory [16], using FPGA-based soft processors.

This paper describes Arvand, a customisable soft processor for fast execution of inductive logic programming. There are four main novelties: (1) an instruction set and microarchitecture for the Arvand processor for executing logic programs (Section 3.1), (2) customisation options for the Arvand processor for improving performance or reducing resource usage to meet requirements for a given application (Section 3.2), (3) evaluation of resource usage (Section 3.3) and performance (Section 3.4) of Arvand, comparing it with general-purpose processors executing large bioinformatics data sets, (4) evaluation of the performance and scalability of multiprocessors based on Arvand (Section 4). Our proposed solution is aimed at exploiting parallelism in inductive logic programming in particular, but many of the techniques can be used for other logic programming-based application domains, such as cognitive robotics [10].

2. BACKGROUND AND RELATED WORK

Inductive logic programming [7] is a learning paradigm based on first-order logic, where learning systems produce predicate hypotheses from background information 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, for example in bioinformatics [12, 13, 15].

Inductive logic programming frames the learning process as a search through a space of hypotheses. We consider in particular the inductive logic programming system Progol, which performs an A*-like search where the quality of a hypothesis is determined by the number of positive and negative examples it correctly classifies as well as its complexity. The searching and testing is computationally demanding, and learning tasks can run for hours or days on modern workstations. To avoid overfitting, i.e. producing a solution which is too specific, cross-fold validation is often used, which increases execution time further. In order to cope with computational complexity, various approaches to parallelising the learning process have been proposed, such as splitting the data set to form hypotheses based on each partition [11], partitioning the language bias [1], and performing a parallel branch and bound search through the hypothesis space [8]. Our approach involves parallelising the quality assessment of hypotheses; it can be used with the other methods mentioned above. The level of parallelism we exploit has a fine task granularity, which can be better exploited in a tightly-coupled single-chip system.

The machine learning system Pro*gol* is based on the logic programming language Pro*log*. We can thus support the execution of Progol by using a processor supporting fast parallel execution of Prolog programs. We do this in our Arvand processor, whose execution model is based on the Vienna Abstract Machine (VAM) [6] execution model for Prolog. Although Arvand can potentially execute arbitrary Prolog programs, inductive logic programming has certain properties which our architecture is designed for; these properties include a limited execution depth for the hypothesis tests (which means that the stack can be reduced and kept on-chip) and that background knowledge often contains a large number of simple facts, which means that mechanisms for handling (and reducing) non-determinacy is important.

Hypothesis

```
active(A) :-
  atm(A,B,c,27,C),
  bond(A,D,E,1),
  bond(A,D,B,7).
```

```
1 h-fstvar A
2 c-goal atm/3
3 g-nxtvar A
4 g-fstvar B
5 g-const c
6 g-int 27
7 g-void
8 c-call
9 ...
```

Background

```
atm(d1,d1_1,c,22,-0.117).
atm(d1,d1_2,c,22,-0.117).
...
bond(d1,d1_1,d1_2,7).
bond(d1,d1_2,d1_3,7).
...
```

```
1024 h-const d1
1025 h-const d1_1
1026 h-const c
1027 h-int 22
1028 h-fix -0.117
1029 c-nogoal
1030 ...
```

(a) Prolog code (b) Arvand instructions

FIGURE 1: Example data from the *mutagenesis* data set. Arvand instructions are prefixed to denote control instruction (`c`), head data instruction (`h`), or goal data instructions (`g`). For both Prolog and Arvand code, upper-case letters denote variables.

Prolog is a declarative language, with an execution model and primitive operations different from those of conventional imperative languages. In particular, Prolog relies on unification for data manipulation, uses tagged data, is strongly stack-oriented, and can have non-deterministic execution. Programs are defined by declaring *facts* and *rules* describing relationships which are held to be true. In the Pro*gol* system, hypotheses are rules, while the background knowledge contains both facts and rules. We use a couple of benchmarks taken from bioinformatics: mutagenesis and protein folding for immunoglobulin. To illustrate how these are represented, Figure 1 shows a hypothesis (top) and a few facts from the background knowledge (bottom) from the mutagenesis benchmark. Column (a) shows these in Prolog form, while column (b) shows the corresponding VAM/Arvand instructions. The hypothesis describes the mutagenic activity of some compound `A` in terms of properties of some of its constituent atoms and bonds. The part of the background knowledge shown lists some such atoms and bonds.

The Arvand instructions are data-oriented; there is a strong correspondence between the Prolog code and the Arvand instructions. Data instructions simply specify the presence of a variable (e.g. `fstvar`, `nxtvar`, `void`), or a constant (e.g. `int`, `const`, `fix`). Control instructions specify the structure of the rule or fact by delimiting its constituent parts (e.g. `goal`, `call`, `nogoal`). The instruction set is high-level as instructions do not directly encode the operations to be performed. There are for example no branching, jump, or stack manipulation instructions, as these operations are implicit in the control instructions.

The VAM execution model uses two instruction streams, head and goal, which are combined at run-time. The head instruction corresponds to the code of a caller, while the goal instruction corresponds to the callee. In Figure 1a, the second line of the hypothesis (corresponding to lines 2–8 in 1b) can be seen as a call, querying the presence of a particular type of atom in compound `A`. To determine this, execution considers possible matches, such as the example background facts shown in the figure. Data from these two code streams are *unified* by being executed pairwise, e.g. Arvand instructions pairs from lines 3 and 1024 onwards. Unification is responsible for parameter passing, returning data, record allocation and field-access in records. Unification of terms can *fail*. When failure occurs, execution backtracks and attempts another path of execution. This non-deterministic execution requires careful stack support in order to reset the processor state.

Our aim is to provide efficient execution of the VAM execution model by parallel execution of multiple processors, exploiting the latest FPGA technology in the implementation. A similar approach is used in existing chip multiprocessors which combine several simple general purpose processor cores tightly coupled together. In addition to providing individual processors customised to the target application, we can also provide an application-optimised processor interconnect.

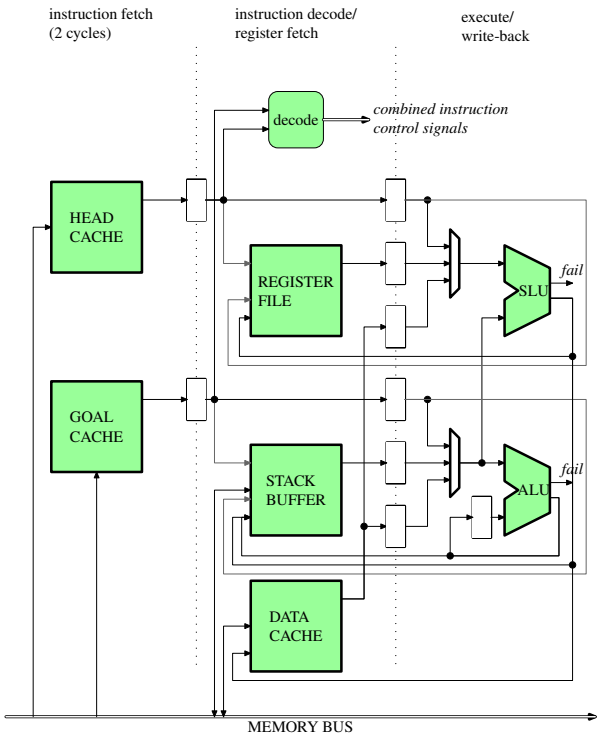

FIGURE 2: Architecture overview of the Arvand processor. The unit labelled SLU is the unifier.

While FPGA-based chip multiprocessors exist [9], they are often not sufficiently optimised for logic programming applications.

We build on recent work on processors for logic programming [2, 3]. Our proposed processor generalises the simple data processor in [2] and supports a much wider range of input (including structured data) than the design in [3]. Both the single processor and the multiprocessor architectures provide a higher degree of microarchitectural customisation than the ones in [3]. While [3] provides only execution time results, we also provide an analysis of the performance benefits of using a high-level instruction set on a tailored architecture.

3. THE ARVAND PROCESSOR

3.1. Processor microarchitecture

The two-pointer VAM execution model is realised as a two-issue pipelined processor (Figure 2). In its most basic form, the processor has four pipeline stages: two-stage pipelined fetch, one decode stage, and one execution stage. There are two instruction pipelines (for head and goal code) which are combined in the execute stage. The two-pointer model increases the control complexity, but supports twice the issue bandwidth. Additionally, the two-pointer model can avoid some redundant computations [6] and tends to generate less data on the heap which in some cases can be eliminated altogether.

The two instruction streams are cached independently in direct-mapped caches. Caches are constructed from small embedded pipelined RAMs with one read port and one write port. Separate caching provides a cheap way of issuing two instructions per cycle. Also, the two instruction streams may well have quite different characteristics, and separate caches allow for changing the design separately. For example, a program dominated by non-deterministic execution may have much higher temporal locality among goal instructions than among head instructions, as the same goal is repeatedly and unsuccessfully unified with different heads.

A set of general-purpose registers contain variables in the current head activation record. Two activation records are active at the same time, however, and the goal activation record is available

in a top-of-stack buffer. Unification operations may require reading or writing to both records, and the processor can do a read and a write each cycle to either record. The reading takes place in the decode stage, while the writing takes place in the execute stage. Forwarding logic avoids read-after-write hazards. There are no global variables, so all data references in the code point into the relevant activation record. References on the stack, however, may point into the heap. Such data references are resolved in the decode stage, where deep reference chains may cause a pipeline stall.

In addition to the general-purpose registers, the processor contains special registers to store activation records. In order to handle backtracking, the stack contains non-determinate activation records (choice points) in addition to the regular activation records (environments), forming two interleaved stacks. The topmost choice point is stored in a special register set in order to speed up backtracking.

The execution stage handles the simple cases of unification as well as arithmetic operations. The unification unit can compare two constant data items, record a binding on either or both of the local stack and register file, and record a fresh unbound variable on the heap. A single simple unification operation, corresponding to two data instructions, can thus be performed every cycle.

There are no explicit stack manipulation instructions, as these are implicit in the complex instructions and backtracking logic. The head activation record is moved between the register set and the stack buffer on call and return. Non-deterministic activation records are pushed on some, but not all, calls, and popped on failure. Stack operations can stall the pipeline, but are partially overlapped with instruction pipeline refills. The implicit stack operations add some control complexity, but reduce the code size. This is useful, as cache sizes on our target platforms are limited.

The processor supports additional operations. These include arithmetic operations, special addressing modes for selecting program paths either based on static information or dynamic information (known as static and dynamic indexing), and support for complex data structures using additional pipeline stages.

3.2. Processor customisation

One advantage of targeting Arvand to FPGAs is that the processor can be customised to different applications. These customisations can have a significant impact on both the resource requirement and performance of the processor. When the processor is used as a processing element in a chip multiprocessor, this results in interesting trade-offs in the aggregate performance, as the number of processors that can fit on the chip depends on the type, and hence performance, of the individual processors.

We focus on three types of customisation: microarchitecture customisations, memory interface customisations, and memory size parametrisations. The microarchitecture and memory interface customisations determine the types of programs that the processor can support, by constraining the valid control and data constructs. This in effect trades off processor generality for resources. These customisations can thus be applied when the characteristics of the target program/data is known, either through automated analysis of a specific program or by user-specified constraints on a class of programs. The memory size parametrisations do not alter the types of programs that are supported, but may affect the performance of each processor. We use the strategy of reducing the processor to the simplest instance that can support the required input programs/data, and then considering trade-offs involving memory size parametrisations. This process can be partially automated [4].

The following microarchitecture customisations are available:

ground: If a program contains only variable-free (ground) terms, a number of data instruction combinations are guaranteed not to occur together, and can hence be eliminated.

unit: If a program contains only facts (unit clauses), rather than rules (except for the initial goal), some control instruction combinations are guaranteed never to occur, simplifying decode and control logic.

indexing: Dynamic indexing instructions, which reduce the amount of non-determinacy at run-time, require support in the form of an extra call instruction, and a special execution mode for traversing a lookup table. For some programs all this information can be determined statically, and the instruction support can be eliminated.

alu: Some programs may rely exclusively on symbolic data and therefore do not require any arithmetic operations (except separately hard-wired address computation) and hence no ALU. When present, the ALU can be customised such that only certain operations are supported.

structures: Unifying structures is supported by a number of different instructions, and requires executing the two pipelines at different speeds. Programs without such structured data can execute without this extra support.

In the most general processor instance, the code, stack, and heap are stored in main memory, and are cached (head/goal instruction caches, data cache) or buffered (top-of-stack buffer) locally on the processor. For some programs, however, these caches and buffers can be configured as purely local units, or can be eliminated altogether. These customisations reduce both the memory resource requirement, and the control logic associated with the buffering/caching. The following memory interface customisations are available:

goal buffer: For programs with the *unit* customisation, the goal cache can be used as a local memory buffer which is loaded during processor initialisation. This avoids overheads in resolving cache misses for the goal instruction stream.

head buffer: For programs of limited size, all the head code can be kept locally.

local stack: For programs whose execution depth is limited, the whole stack can be kept locally.

no heap: The heap stores bindings for structured data and new unbound variables. Programs which do not have either structured data or the relevant variable unifications, can execute on a program without a heap.

local heap: For programs whose execution depth is limited, the data can be kept locally, instead of being cached.

The mutagenesis benchmark can execute on a processor using the ground and unit customisations, without dynamic indexing, heap, or structure support. Our other benchmark, immunoglobulin, can execute on a processor with an ALU and dynamic indexing support, but without heap or structure support.

3.3. FPGA implementation and trade-offs

Resource constraints is a major issue for our soft processor implementation. We assume a generic target FPGA with configurable logic blocks (CLBs) and embedded memories. Either of these resources can pose a limiting factor. While simple processor configurations use few resources, adding features can significantly increase the programmable logic cost. Programs which can operate using a reduced instruction set and limited amounts of memory are therefore the most suitable for exploiting the chip multiprocessors that we propose.

We have implemented the Arvand processor on the Xilinx Virtex II chip XC2V6000, which has a 0.15/0.12μm feature size. The XC2V6000 chip contains 6M system gates, divided into 8448 configurable logic blocks (CLBs), as well as 144 embedded 18Mb RAMs. Virtex II CLBs are based around eight 4-input look-up tables (4-LUTs). Area cost can be separated into CLB cost for

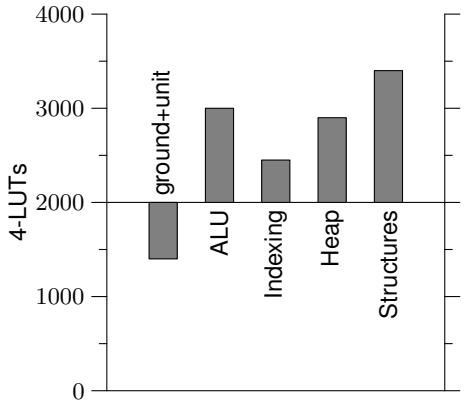

FIGURE 3: Approximate resource cost/benefit of customisations of the Arvand processor measured in terms of device-independent 4-input lookup tables (4-LUTs), relative to a single processor in the "normal" configuration with two small 512-word caches.

programmable logic and embedded RAM cost. Since CLBs are architecture specific, we report the cost in terms of 4-LUTs.

Figure 3 shows the approximate cost of the different microarchitecture customisations using the XC2V6000 device, measured relative to a single processor in the "normal" configuration (no customisations). Processors can use several of the given customisations, which have an additive effect on the total resource usage. There can therefore be a significant difference in resource requirement between the simplest and most complex configuration.

Memory units are composed from embedded RAMs. We consider composing up to 16 embedded RAMs configured as 512×36 bit units, resulting in a range of memory unit sizes from 512 words to 8K words. An 8K-word memory unit uses 11% of the available memory resources on the XC2V6000 chip. Additionally, there is overhead in terms of the connective logic required when composing embedded RAMs. Thus a restriction of current technology is the limited size of caches and buffers if the processor is used in a chip multiprocessor.

3.4. Single processor performance

The Arvand instruction set contains complex instructions targeted specifically to the Prolog/VAM execution model. The number of instructions that needs to be executed for a given program is therefore smaller for Arvand than for a general-purpose processor (GPP). On the other hand, targeting FPGAs results in a slower clock. Also, the more complex instructions are likely to require more cycles per instruction (CPI).

Executing Prolog on a general-purpose processor is often carried out via an abstract machine, where the abstract machine instructions are either interpreted in a byte-code interpreter or are used to guide compilation to the native instruction set. In contrast, the Arvand processor directly executes such abstract machine instructions. Since the abstract machine instructions are more complex than native microprocessor instructions, there is an interpretation overhead, O_{int}, which is the number of native instructions required to execute each abstract machine instruction.

Arvand is faster than a general-purpose processor when its reduced interpretation overhead outweighs its dual disadvantages of lower clock frequency and higher CPI. The speedup of Arvand over the equivalent software executing on a GPP is given by: $speedup = O_{int} \times \frac{CPI_{GPP}}{CPI_{Arvand}} \times \frac{f_{FPGA}}{f_{GPP}}$ where typically the clock frequency ratio f_{FPGA}/f_{GPP} is in the range $1/10$ to $1/40$, $CPI_{GPP} < 1$, and $CPI_{Arvand} > 1$. This is a simplification as it ignores the effect the memory architecture, necessarily different on the FPGA, has on performance.

We compare the FPGA implementation of Arvand, $\text{Arvand}_{\text{FPGA}}$, with three software Prolog systems: YAP Prolog, a fast direct-threaded byte code interpreter; GNU Prolog, a native-compilation system; and $\text{Arvand}_{\text{SW}}$, a software implementation of the Arvand execution model. We use the $\text{Arvand}_{\text{SW}}$ target since $\text{Arvand}_{\text{FPGA}}$ follows a different basic execution model from YAP, and uses small statically sized memory segments. For the benchmarks in this paper, $\text{Arvand}_{\text{FPGA}}$

Benchmark	System	O_{int}	CPI_{GPP}	CPI_{Arvand}	Speedup of Arvand at 40MHz	Speedup of Arvand at 100MHz
mutagenesis	$Arvand_{SW}$	34	1.38	2.05	0.37	0.92
	YAP	82[b]	1.2[b]		0.41	1.03
	GNU Prolog	33/60[a]	1.01		0.25	0.64
immunoglobulin	YAP	84[b]	1.2[b]	1.96	0.96	2.40
	GNU Prolog	45/39[a]	1.38		0.51	1.28

[a]Based on Arvand/YAP abstract machine instruction counts
[b]Based on estimated CPI value

TABLE 1: Performance comparison between a single $Arvand_{FPGA}$ and three software Prolog systems: $Arvand_{SW}$, YAP Prolog, and GNU Prolog. The interpretation overhead, O_{int} is the number of native instructions required to execute one abstract machine instruction, based on the CPI in the next column. The speedup column shows the speedup of $Arvand_{FPGA}$ over the given software when $Arvand_{FPGA}$ is clocked at 40MHz and 100MHz. The larger this number, the larger the advantage of Arvand over software.

places data exclusively on the stack, and hence does not perform garbage collection. $Arvand_{SW}$ is a bytecode interpreter which has an identical execution path as $Arvand_{FPGA}$ and executes $Arvand_{FPGA}$ binaries.

We execute the software Prolog systems on a 2.53GHz Pentium 4 (Northwood) processor, which is based on comparable technology to the Virtex II FPGA. The $Arvand_{FPGA}$ results do not consider variations in cache or other architectural characteristics, and are based on a 2K-word head instruction cache and a 1K-word goal instruction cache. The current implementation of Arvand on the XC2V6000 executes at 40MHz. We also make a comparison with the processor executing at 100MHz, which we consider a realistic clock frequency after further optimisations.

We consider two benchmarks taken from bioinformatics: mutagenesis and protein folding. The Mutagenesis data set is a machine learning benchmark from a study [12] of the mutagenic activity in nitro-aromatic compounds, linked to cancer. It contains simple program constructs and relies heavily on backtracking. *mutagenesis* is a sample of 200 hypothesis tests from running this benchmark through Progol. The mutagenesis benchmark uses 46KB of program data.

The *immunoglobulin* benchmark is taken from an inductive logic programming application for detecting protein fold signature data [14]. The data set refers to different protein folds and is sampled from the hypothesis tests relating to the immunoglobulin protein. The data set requires the full range of control instructions and includes a modest amount of backtracking. There are some arithmetic operations but no structures. The *immunoglobulin* benchmark uses 157KB of program data. This data set can be executed on a processor with ALU and dynamic indexing support only, without heap or structure support.

The speedup of $Arvand_{FPGA}$ over software is between 0.25 and 0.96 when $Arvand_{FPGA}$ is clocked at 40MHz, i.e. at its best the same performance as the software. When $Arvand_{FPGA}$ is clocked at 100MHz, its performance is correspondingly better, with speedups in the range 0.64 to 2.40. For the purpose of measuring the CPI of Arvand, a pair of instructions (head + goal) is counted as *two* instructions, even though they are executed as a combination. The theoretical minimum CPI for Arvand is thus 0.5. The measured CPI for Arvand is around 2, a result of the high cost of control hazards. The performance of Arvand with respect to the software targets is better for *immunoglobulin* than for *mutagenesis*. This can be attributed to the relatively higher cost of backtracking compared with unification in Arvand. The mutagenesis benchmark is dominated by backtracking. The interpretation overhead is lower for GNU Prolog than for YAP, which can be expected, since it compiles directly to the native instruction set. This estimated overhead is consistent with the assembly code generated by the GNU Prolog compiler.

Overall, we find that despite the higher CPI and lower clock frequency, a single Arvand processor only has a slight disadvantage with respect to software running on a general-purpose processor. This is achieved by a much reduced instruction count and a customised microarchitecture. It thus forms a good basis for chip multiprocessor implementations.

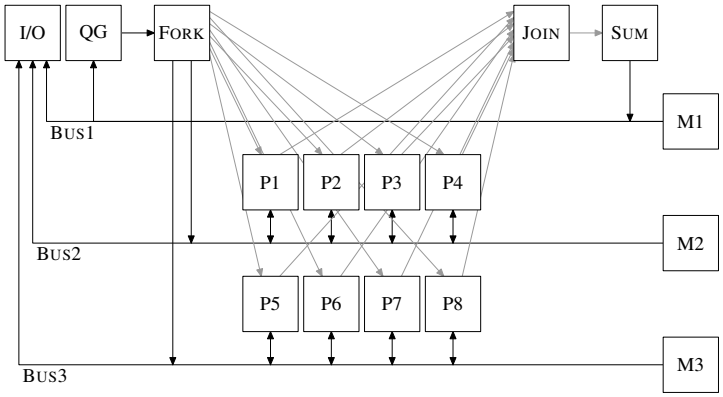

FIGURE 4: Multiprocessor structure for Progol multiprocessor. Each box labelled P_i denotes an instance of the Arvand processor

4. PROGOL MULTIPROCESSOR

The computational kernel of Progol can be executed using an array of processors to evaluate the stream of hypotheses. Figure 4 shows the structure of such a Progol multiprocessor with eight independent processor cores (P_1–P_8). In practice the precise number of cores is typically larger and depends on the processor configuration and the target device.

The input to this Progol-specific multiprocessor consists of: (a) a logic program encoding the background knowledge, (b) a stream of program snippets encoding the hypotheses, and (c) a list of examples. The background knowledge is written to the shared code areas on each of the memories used by the processors (M_2 and M_3), while the hypotheses and examples are written to input buffers in M_1. The hypothesis test is produced on-chip by combining a hypothesis with each of the examples in a query generator (QG). The resulting stream of program calls are dispatched to available processors by a FORK unit which writes the program call data to each processor's buffer in M_2 or M_3. The results are synchronised by the JOIN unit and accumulated by the SUM unit, and then written back to the output buffer. The FORK and JOIN units are pipelined to avoid becoming a critical path due to high fan-in and fan-out.

To evaluate the scaling properties of the multiprocessor, we use the processor customisations for the two benchmarks outlined in Section 3.2. For *mutagenesis* we use up to 16 processors on the XC2V6000 using a fixed goal cache size (512 words), and varying the head cache size between 512 words and 4K words. For *immunoglobulin* we use up to 8 processors with different microarchitectural customisations, but otherwise the same setup. The speedup in each case is measured relative to a single processor with a 512-word head instruction cache. As bus congestion can be an issue for larger number of processors, we additionally find the average, maximum, and minimum utilisation for each of four banks for the different cache configurations.

For *mutagenesis* the speedup is near-linear for up to 8 processors (Figure 5a). For 16-processors the speedup falls somewhat, indicating that memory bus congestion and dispatch overhead becomes significant. There is no noticeable difference between the different cache configurations, indicating that there are few capacity misses, even for the smallest caches. For this reason, the external memory utilisation (Figure 5b) is similar for the different cache configurations, but increases to almost 50% for 16 processors.

For *immunoglobulin* the speedup is near-linear for up to 8 processors (Figure 6a). The larger cache configuration (4K words) shows a 12% higher performance than the smallest (512 words). The difference in memory utilisation for the different cache configurations (Figure 6b) results in differences in speedup, but the effect is not large for 8 processors, as the total memory utilisation is low, at 25%.

The number of nodes in the multiprocessor can be varied, along with the configuration of each processor. When scaling the number of nodes in the multiprocessor, there are two resource limitations: the logic required for the processor and the memories required for its buffers and caches. There is thus a trade-off where increasing the number of processors limits the amount

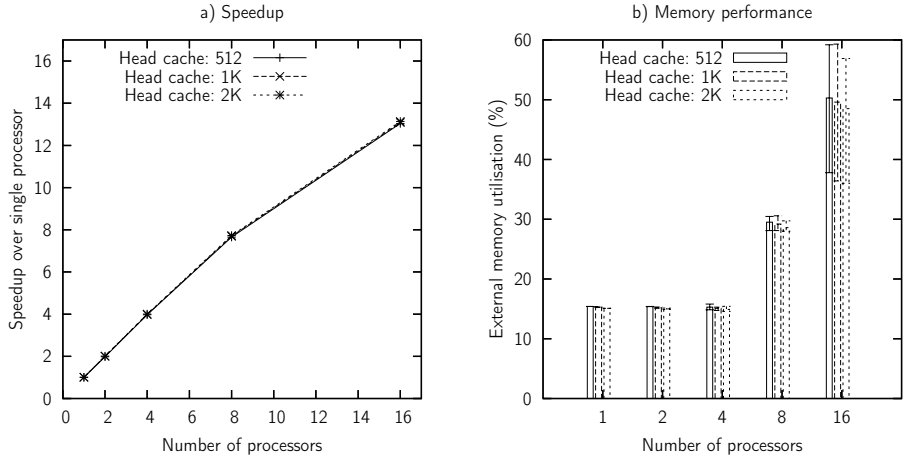

FIGURE 5: Speedup and memory utilisation of the *mutagenesis* benchmark for different cache configurations of the Progol multiprocessor.

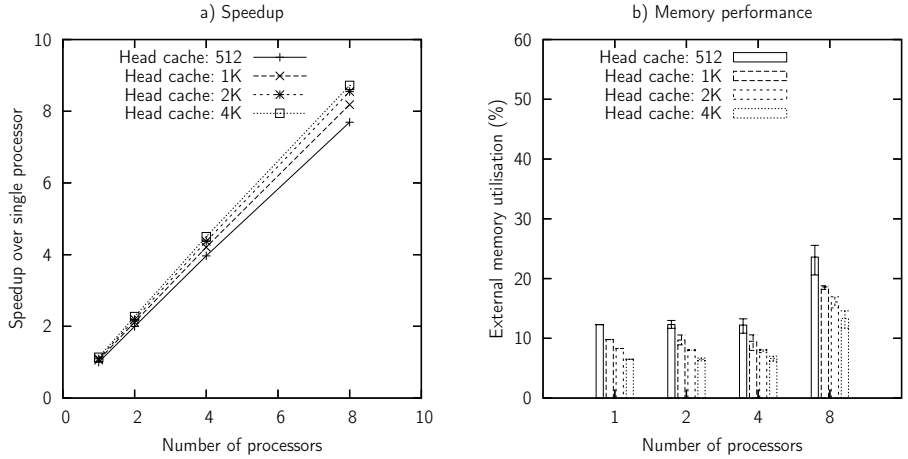

FIGURE 6: Speedup and memory utilisation of the *immunoglobulin* benchmark for different cache configurations of the Progol multiprocessor.

of memory available to each processor. Figure 7 shows this trade-off for (a) the XC2V6000 chip (based on place-and-route results), and (b) the larger XC4VLX160 chip (estimated).

The configuration used for the *mutagenesis* benchmark allows up to 32 processors on a chip for small memory sizes, but this number quickly drops when the total memory size increases. The configuration used for *immunoglobulin* allows a smaller number of processors, 14 on the XC2V6000, but this means that a larger total memory size is required before the multiprocessor placement becomes memory resource bound. Figure 7b shows a similar trend for the larger XC4VLX160 FPGA, which should fit 65 processors for *mutagenesis* and 30 processors for *immunoglobulin*.

We find that up to 32 processors for the *mutagenesis* data set can fit on the XC2V6000 chip. A 16-processor configuration gives a speedup of around 13 with respect to a single Arvand processor. Extrapolating to 32 processors, and taking bus utilisation into account, we expect a speedup of around 20 times compared with a single Arvand processor. Combined with the single-processor results for *mutagenesis* the total FPGA speedup with respect to GNU Prolog running on a 2.53GHz Pentium 4 is then around 5 times when Arvand is running at 40MHz and around 13 times when it is running at 100MHz. Doing the same analysis for *immunoglobulin*, a 14-processor configuration would have a speedup of around 6 times when running at 40MHz and 15 times when running at 100MHz.

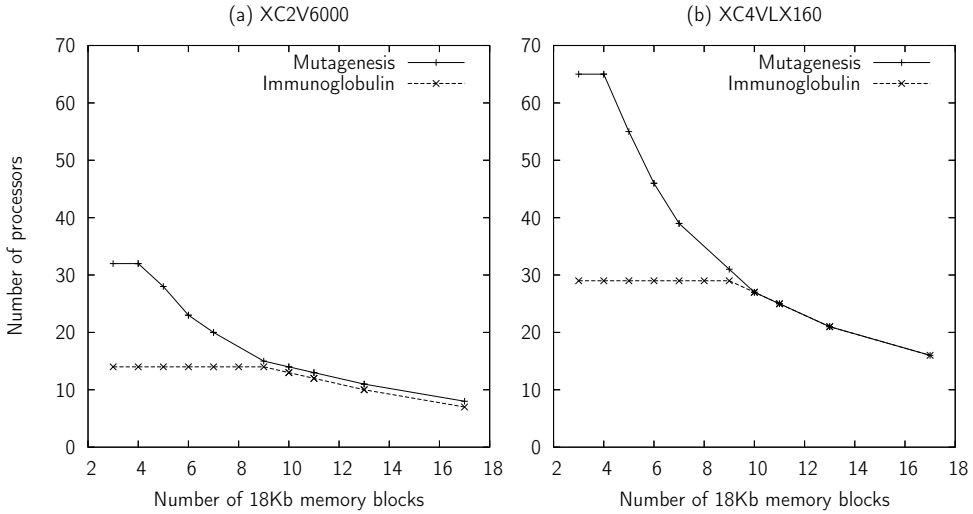

FIGURE 7: Trade-off between number of processors and memory usage per processor for the Progol multiprocessor on two types of FPGAs: (a) XC2V6000, (b) XC4VLX160.

5. CONCLUDING REMARKS

This paper presents the Arvand processor and multiprocessor architectures for accelerating machine learning in the inductive logic programming domain. Prototype implementations on XC2V6000 and XC4VLX160 FPGAs show the viability of the approach: that a combination of complex instruction set, domain specialisation and application customisation can give better performance than general-purpose processors. The low resource usage of the processor allows multiple instances to fit on a chip, and the machine learning application readily supports exploitation of this form of parallelism.

The Arvand design can be further improved in several ways. For instance, soft processors on the same FPGA device have been reported to be capable of 170MHz [17], while our design is currently clocked at 40MHz. Even if the current design cannot be pushed that far, the memory bandwidth can be better utilised and can be clocked faster than the processor. A four-time performance improvement appears possible through a combination of higher processor and memory clock speeds. Space usage could also be improved, perhaps by a factor of two. Moreover, there is room for microarchitectural improvements in the processor, especially with regards to branch overheads, which might yield a performance benefit of 50%-100%. We therefore hope that, using current technology, the performance of Arvand can be improved by a factor of ten, to achieve an overall speedup of up to two orders of magnitude over general-purpose processors.

Compiler optimisations can have a significant impact on overall performance. We have factored compiler optimisations out of the comparisons between the Arvand processor and the general-purpose processor for a fair comparison of the raw computational power in this application domain. To some extent compiler optimisation is an orthogonal issue, as such optimisations can be applied to both general- and special-purpose processors. However, a highly specialised processor can support hardware optimisations which may incur fixed costs in software. An example, not described earlier, involves an Arvand optimisation which trims non-deterministic branches of execution at run time by setting a flag in one instruction, giving a speedup of 40 times for the *mutagenesis* data set. This has only a small hardware cost and no run-time cost in cases where it does not apply. In contrast, implementing the same feature in software, especially for dynamically generated code as in Progol, could incur additional run-time overhead to decode the flag even where the optimisation is not used as the flag is read. The results in Section 3.4 and Section 4 do *not* use this optimisation for Arvand.

This paper compares the Arvand processor with a conventional single-core microprocessor. The recent trend, however, is towards multicore processors. Such multicore processors can clearly

exploit the same type of parallelism in Arvand. However, Arvand has been shown to be competitive with current microprocessors in comparable technology. Future research would explore whether the customisability of Arvand, coupled with the growth in FPGA capacity, would continue to make Arvand-like processors an appealing option. Further work would also consider improving the design to attain higher clock rates, making optimal use of the on-chip memory bandwidth, and exploiting additional forms of parallelism [1, 8, 11]; recent advances in nanowire-based programmable logic [5] could offer a long-term promise for this approach.

ACKNOWLEDGEMENTS

The support of UK EPSRC (EP/C51050X/1 and EP/C549481/1), the ORS Award Scheme, the HiPEAC project, Agility, Celoxica and Xilinx is gratefully acknowledged.

REFERENCES

[1] Dehaspe, L. and De Raedt, L. (1995) Parallel inductive logic programming. *Proc. MLnet Familiarization Workshop on Statistics, Machine Learning and Knowledge Discovery in Databases*, Heraklion, 112–117.

[2] Fidjeland, A. and Luk, W. (2003) Customising parallelism and caching for machine learning. *Proc. IEEE Int. Conf. on Field-Programmable Technology*, Tokyo, 204–211.

[3] Fidjeland, A. and Luk, W. (2005) Customising application-specific multiprocessor systems: a case study. *Proc. IEEE Int. Conf. Application-Specific Systems, Architectures, and Processors*, Samos, 239–244.

[4] Fidjeland, A. and Luk, W. (2006) Archlog: High-level synthesis of reconfigurable multiprocessors for logic programming. *Proc. Int. Conf. Field-Programmable Logic and Applications*, Madrid, 335–340.

[5] Gojman, B. et al. (2006) 3D Nanowire-based programmable logic, *Proc. Int. Conf. on Nano-Network*, Lausanne, 1–5.

[6] Krall, A. and Neumerkel, U. (1990) The Vienna Abstract Machine. *Proc. Int. Workshop Prog.Language Implementation and Logic Programming*, LNCS 456, 121–135.

[7] Muggleton, S. and De Raedt, L. (1994) Inductive logic programming: Theory and methods. *J. of Logic Programming*, 19,20:629–679.

[8] Ohwada, H. et al. (2000) Concurrent execution of optimal hypothesis search for inverse entailment. *Proc. Int. Conf. Inductive Logic Programming*, 165–173.

[9] Ravindran, K. et al. (2005) An FPGA-based soft multiprocessor system for IPv4 packet forwarding. *Proc. Int. Conf. Field-Programmable Logic and Applications*, 487–492.

[10] Shanahan, M.P. and Randell, D. (2004) A logic-based formulation of active visual perception. *Proc. Int. Conf. Principles of Knowledge Representation and Reasoning*, 64–72.

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

[12] Srinivasan, A. et al. (1994) Mutagenesis: ILP experiments in a non-determinate biological domain. *Proc. Int. Inductive Logic Programming Workshop*, Gesellschaft fur Mathematik und Datenverarbeitung MBH, GMD-Studien Nr 237.

[13] Sternberg M. and Muggleton, S. (2003) Structure activity relationships (SAR) and pharmacophore discovery using inductive logic programming (ILP). *QSAR and Combinatorial Science*, 22.

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

[15] Turcotte, M., Muggleton, S. and Sternberg, M. (2001) Automated discovery of structural signatures of protein fold and function. *J. of Molecular Biology*, 306:591–605.

[16] Wee, S. et al. (2007) A practical FPGA-based framework for novel CMP research, *Proc. Int. Symp. on FPGAs*, Monterey, CA.

[17] Xilinx. (2006) MicroBlaze Soft Processor – Performance, Xilinx Inc., CA.