Instruction-level Parallelism in Prolog: Analysis and Architectural Support

Alessandro De Gloria and Paolo Faraboschi

University of Genoa
Department of Biophysical and Electronic Engineering (DIBE)
Via Opera Pia 11a
I-16145 Genova, Italy
e-mail: adg@cpsi7&be.unige.it

ABSTRACT

The demand of increasing computation power for symbolic processing has given a strong impulse to the development of ASICs dedicated to the execution of Prolog. Unlike past microcoded implementation based on the Warren Machine model, novel trends in high performance Prolog processors suggest the implementation of RISC-based processors committed to Prolog only through the adoption of a few basic dedicated features, like the Berkeley Abstract Machine (BAM) processor.

Following the idea of using a smart compiler for a simple instruction set, the SYMBOL project represents an experiment in applying global compaction techniques and VLIW design philosophy to the static exploitation of instruction-level parallelism in Prolog.

This paper presents code analysis results and shows how we can approach the theoretical speed-up limit (about 3) imposed by Amdahl's law on shared memory models, by means of global code optimizations and a suitable architectural support. In addition, we show implementation details and some preliminary data of a VLSI prototype architecture.

Keywords: Prolog shared-memory model, Static Instruction-Level Parallelism, Global Compaction Techniques, VLIW architectures.

1. INTRODUCTION

Logic programming, and Prolog in particular, has become a subject of much attention, as it can map complex problems into machine code more easily than traditional high-level languages [Clocksin81]. This is accomplished by relieving the programmer from the implementation of the control structures underlying the algorithms which are provided by the language engine itself [Kowalski79].

Unfortunately the necessity of handling tagged data in Prolog imposes heavy limits on general-purpose machine performance, since uncommitted datapaths lack in dedicated structures and need to execute complex mask constructs for simple operations, like tag insertion or branching.

Unlike early Prolog machines, which implemented microcoded interpreters of Warren Instructions [Warren83], such as the PLM [Debry87] or the KCM [Benker89], novel approaches are based on RISC-based processors and sophisticated front-end compilation techniques, like the Berkeley Abstract Machine (BAM) [Holmer90]. By the adoption of data-flow and mode analysis and specialized unification, the BAM succeeds in exploiting the determinism in Prolog programs and eliminates all the redundancies of Warren Code, thus obtaining an impressive speedup with respect to previous WAM implementations.

This research aims at evaluating quantitatively the maximum benefit which can be obtained by adopting a compiler-driven VLIW architecture [Fisher83] for the BAM model of Prolog. The underlying idea is to utilize a simple load-store architecture with the minimum amount of language specific hardware features and to determine which is the degree of statically exploitable instruction-level concurrency in Prolog.

By applying sophisticated compilation techniques to simple hardware with only a limited application specific support, our approach benefits from the fine-grain parallelism of code, and at the same time does not compromise general-purpose computation capabilities and flexibility.

This paper presents the results of code analysis performed on Prolog benchmarks extracted from the Aquarius Benchmark Suite [Haygood89], and how they justify the adoption of global compilation techniques, which were originally developed for numeric code [Fisher81]. In section 3 we also sketch a description of the evaluation system and the code-compaction mechanisms we implemented in the back-end compiler. Finally, section 5 presents some preliminary data of a prototype VLSI processor which has been designed and is currently being tested in its operating environment.

2. THE BERKELEY ABSTRACT MACHINE

The Berkeley Abstract Machine (BAM) has been developed as part of the Aquarius Project at University of California, Berkeley [VanRoy90]. The BAM execution flow and data structure is similar to the Warren Abstract Machine (WAM) [Warren83], but it uses an instruction set which is much closer to a general-purpose architecture. This allows a higher possibility of low-level optimizations than the WAM. BAM code has been designed to be re-targetable to a wide range of architectures, by the adoption of an instruction set which is machine and implementation independent.

Some basic guidelines have been followed in the design of the BAM model:
- Development of techniques to exploit determinism in programs
- Development of techniques to specialize unification
- Development of a portable instruction set suitable for optimizations

The basic idea of the BAM is to retain the good features of the WAM, but implementing an instruction set closer to RISC-based architectures. In addition, particular attention has been focused on unification, which has been specialized by means of a

© 1992 ACM 0-89791-509-7/92/0005/0224 $1.50
global dataflow analysis which derives information about the program structure at compilation time.

A first implementation of a BAM processor [Holmer90] reaches performance which are approximately a factor of ten faster than Warren-based implementation like the PLM [Dobry87]. The factor of ten comes from:

- a model improvement of the BAM versus the WAM abstract machine and more sophisticated compiler optimizations. This gives roughly a factor of three.
- a technological improvement in VLSI processor design. This allows a clock rate (30 MHz) about three times faster than the microcoded PLM (10 MHz).

3. THE EVALUATION SYSTEM

In order to quantitatively evaluate the data-path resources which can actually bring cost-effective performance gain to the BAM model, we have applied some general criteria which are valid for any domain specific architecture [DeGloria90].

The development of a domain specific architecture requires analysis and evaluation tools to guide the designer towards correct design choices, and must show some basic features:

- the starting point of the design must be the behavior of the architecture, specified by means of benchmarks which has to be free as much as possible from hardware hypotheses;
- the minimum set of hardware functionalities being able to speed up the execution of the application must be found out;
- a resource has to be introduced only if it is justified by its frequency of use and if it can be efficiently exploited by the available compiler technology.

Our analysis considers the class of parallel synchronous non-homogeneous architectures which we define through the following properties:

- There is one central control (one program counter).
- The Data Path is composed of several functional units.
- The functional units are connected by means of local or global buses.
- The functional units may work in pipeline or multi-cyclic mode.
- The machine instruction may be composed of several operations, each of them driving a different functional unit.
- Each functional unit requires a static predictable amount of execution time.
- A new instruction is issued at each machine cycle.
- Memory can be organized in a hierarchical fashion.

This architecture model represents a large set of architectures which encompass VLIW machines. No constraint is imposed on the complexity of the functional units, and the only restriction to the model is to be synchronous in order to manage parallelism in a static way.

During evaluation, hardware and software are considered tightly coupled components of the design process, and both architecture and compiler features can be analyzed. The evaluation system (Figure 1) starts from the program benchmarks and allows to obtain a set of measurements of various architecture configurations through a sequence of intermediate steps.

3.1. Front-end and Intermediate Code features

The front-end implies the translation of the Benchmarks into an Intermediate Code (IntCode), composed of simple instructions directly expressing primitive hardware functionalities. This is accomplished by means of two steps: Prolog compilation and Intermediate Code translation.

- We compile Prolog into the BAM abstract machine model by using the BAM compiler [VanRoy90] with the Automatic Mode analysis option in order to get high-performance code.
- BAM code is translated into machine-independent Intermediate Code Instructions (ICI). In this step we avoid all optimizations which are delayed to the back-end compiler. We only apply a variable renaming procedure in order to eliminate redundant data-dependencies.

The IntCode is composed of simple instructions directly expressing functionalities which, for the specific application, represent a good hardware/software tradeoff. Each ICI is directly executed by a machine hardware resource. ICIs specify functionality but contain no information about register allocation or hardware units. The general format of an ICI is:

\[ \text{OpCode} \text{ Operands; } \ldots, \text{ Operands} \]

where the operands may be variables or constants. Only direct and immediate addressing modes are supported while indirect addressing must be detailed with more instructions.

![Diagram](image-url)
An IntCode Sequential Emulator extracts basic statistical information about the code generated by the front-end from Prolog Benchmarks, and validates the code itself. The emulator computes for each IC the Expect (number of times it has been executed), and for each branch also the Probability (to take the branch). These information are used by the Parallelizing Compiler to guide global compaction.

3.2. Back-end Compilation

The back-end compiler extracts the parallelism from the IntCode and maps it onto the resources of a target architecture.

It is based on a global parallelizing technique derived from the Trace Scheduling approach [Fisher81], which operates compaction on traces of instructions extending beyond each basic block of code. Trace choice is based on the statistical information about execution frequency extracted by preliminary simulation. Any instruction of any complexity can be modeled, with the only constraint that each operation must require a fixed and known amount of time to be executed.

The Code generator is implemented by means of a variation on the Bottom-Up-Greedy algorithm described in [Ellis85].

Functional units for operations and register are chosen with heuristic techniques which take into account:
- estimated completion cycle of an operation for each possible unit in the architecture;
- nearest allocated operands and destinations;
- possible bottleneck paths, using a priority assigned to each communication path;
- register movement insertion in order to have operands and destinations directly connected with the functional units;
- functional unit usage and register allocation balance;

An event-driven simulator has also been developed in order to obtain performance and statistics about resource utilization.

4. CODE ANALYSIS

Some important remarks have to be done about the evaluation methods.
- The analysis focuses only on the micro-architecture features without considering system related parameters. On this purpose, we assume to have a fixed access time for both data and code cache memory. The independency of the micro-architecture from other system components enables to scan a space of homogeneous micro-architectures.
- We intentionally avoid any reference to absolute cycle timing, highlighting technology-independent parameters. What we want to estimate is the performance/cost gain that can be obtained with the introduction of different architectural options in comparison with a pure sequential implementation. Absolute values (cycle time, chip area) are related only to available technology and have no influence on the presented analysis.

4.1. Data dependency and memory references

Dependency analysis is fundamental for parallelism discovery: it allows the compiler to exploit fine-grain instruction concurrency by transforming the program in a semantically equivalent form with a higher degree of parallelism. Dependency analysis is necessary in particular for memory disambiguation for parallel access on separate memory banks [Wolfe89]. The problem of data-dependency reduction is then outstanding from a parallel perspective, and in the numeric world a wide set of optimizations algorithms has already been developed [Aho86].

In this context, it is important to remark that the Abstract Execution Model of the BAM (as well as the WAM [VanRoy84]), separates data space in distinct stack areas (environment stack, choice point stack, heap, trail, push-down). Then it would seem natural to physically organize data cache memory in different banks in order to have an easy parallel memory access. Unfortunately, previous studies on the WAM [Tick87], which retain their validity for the BAM as well, have demonstrated that the majority of memory accesses are in the stack (environment and choice point: more than 70%) and cannot be disambiguated since most of the times deriving from an indirect (pointer) memory address. Since no valid technique has yet been developed for pointer disambiguation, memory references analysis cannot help to eliminate data-dependencies.

Moreover, optimizations which are commonly employed in numeric code, such as loop unrolling and interchanging, are nearly useless in Prolog due to the peculiar nature of symbolic computation. In fact, Prolog code is characterized by an absence of "traditional" loops, the only ones being built-in predicates (e.g. multiplication, division, hashing), dereferencing and detracling, while iterative constructs are normally implemented through recursion. Since unrolling is mainly used to eliminate memory dependencies in array-based structures, it is completely ineffective in Prolog loops which basically perform pointer-chasing operations.

It is important to remind that distributed memory models for Prolog have already been developed in the past, although they have not had wide popularity, since in many cases the semantic of the language was altered, or the programmer was forced to specify program parallelism.

For these reasons we limit our research scope to shared-memory architectures, although the same compilation methods and algorithms can be easily extended to multiple memory access configurations, when suitable models will be developed.

4.2. Instruction Frequency and Amdahl's Law

Prolog programs spend a large part of time in data memory accesses. Measurements on instruction frequency on a subset of programs extracted from the Aquarius Benchmark Suite (Figure 2) show that memory operations take about 32% of the whole execution time. Data are computed as an average of the values obtained via sequential simulation of the benchmarks and with the hypothesis that all operations have the same duration.

We can apply well-known Amdahl's law [Amdahl67] to evaluate which is the maximum speed-up we can expect with a shared memory approach:

$$\text{Speedup} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) \cdot \text{Fraction}_{\text{enhanced}}}$$

If we suppose to speed up ideally the execution of all operations but memory accesses, we can say that:

$$\text{Fraction}_{\text{enhanced}} = 0.68$$

$$\text{Speedup}_{\text{enhanced}} \rightarrow \infty$$

$$\text{Speedup} = \frac{1}{0.32} = 3.0$$

As a consequence, with a shared memory model, we will never be able to obtain a speed-up of more than 3.0 over a sequential implementation. Our opinion is that a value of 3.0 as an asymptotic speed-up is not to be considered marginal and can justify the research by itself. Obviously, we cannot speed-up ALU operations, data movements and control operations without limits, but we have to take into account implementation cost issues. Figure 3 gives an approximation of the maximum ideal speed-up which can be obtained by varying the enhancements of ALU and control operations. The dotted line shows the maximum hypothetical speed-up when supposing that memory operations are executed separate from computation. The continuous line, on the contrary, is based on the assumption that memory operations can be com-
pletely overlapped with computation. In the latter hypothesis, we can see how factors of concurrency greater than three are useless, since memory accesses constitute the limiting factor.

In this context, it is important to remark that the graph shows an optimistic prevision of speed-up based on the hypothesis that:
- the code contains enough parallelism by itself
- the compiler is able to find the necessary concurrency in the code.

In the following sections, we show how this goal can be reached only by using global compaction techniques which extend their analysis beyond basic blocks of code.

4.3. Basic Blocks and Global Compaction

As we have seen, Prolog code promises a worthwhile degree of concurrency even for a shared memory architecture model, at the condition that an efficient compiler is able to exploit it.

The greatest problem in finding parallelism in Prolog code comes from the high percentage of branch operations (more than 15%) which make difficult to find reasonable degrees of instruction concurrency inside each basic block (composed of not more than 6-7 instructions).

To overcome the barrier of basic blocks, it is then necessary to apply Global Compaction techniques which extend their optimizations beyond basic blocks and consequently allow a higher possibility of code motion. An in-house arrangement of the “Trace Scheduling” technique has been chosen for the analysis, thus allowing to exploit parallelism beyond branches.

Quantitatively, Table 1 shows the available concurrency for each of the considered benchmarks, by limiting code compaction within basic blocks, and by considering global compaction applied to traces of instructions. It is possible to see how a global optimization technique increases the block size to 11-12 instructions, and performs about 30% faster than simple basic-blocks optimizations.

Hypotheses of measurements:
- All operations have duration 1, except memory (2 cycles in pipeline) and control (2 cycles in pipeline). This assumption is made for a realistic modeling of RISC-based machines.
- While computing maximum speed-ups, different kinds of data dependency have been considered: Memory dependency, Source-Destinations, Write-after-read, Write-after-write, Off-live. Also a constraint on the sequence of branches has been imposed in order to limit the possibility of code motion to avoid an exponential growth of instruction copies [Ellis85].
For these reasons, Trace Scheduling has been applied in the past only to highly regular scientific code. In loop-structured code it is easy to predict which direction is the most frequently followed at each branch instruction. In fact, most backward going branches are loop branches, whose typical probability to be taken is about 90%. Starting from these considerations, a well-known rule-of-thumb is derived, stating that about 90% of backward-going branches are taken while only about 50% of forward-going branches are taken [The 90/50 Branch-Taken Rule] [Hennessy90]. This rule has been proved to be rather accurate for typical numeric/scientific code. If it were valid for Prolog code as well, the applicability of Trace Scheduling would be compromised, since the behavior of program execution flow would result unpredictable (branch probability around 50%), due to the absence of loops.

In order to evaluate the Predictability of Prolog code, we have measured the actual distribution of branch probability in the benchmarks.

We can define:
- Probability of a Branch \( P_{br}(b) \): the probability of a branch to be taken,
- Probability of a Faulty Prediction of a Branch \( P_{fp}(b) \): the probability that a branch usually taken is not taken, or that a branch usually not taken is taken. In formulas:

\[
P_{fp}(b) = \left\{ \begin{array}{ll}
P_{bd}(b), & \text{if } P_{bd}(b) \leq 0.5 \\
1 - P_{bd}(b), & \text{if } P_{bd}(b) > 0.5
\end{array} \right.
\]

During Trace Scheduling compilation, we pick the most probable traces of instructions by following the most probable executed branches. When the execution follows the most probable traces, performance gain is maximal. Otherwise, a penalty is paid in terms of additional cycles. Then the average value of \( P_{pd}(b) \) represents a measure for the effectiveness of a Trace Scheduling approach: the smallest \( P_{pd}(b) \), the smallest the probability and the penalty of making a wrong choice during trace picking.

Table 2 shows the average values of \( P_{pd}(b) \) obtained through a dynamic analysis during simulation which computes an average of the probability weighted with the execution frequency of the branches. Figure 4 shows the distribution of \( P_{pd} \) for all the considered benchmarks. We can notice how the average probability of a faulty prediction is about 0.14, value which guarantees a low performance decay due to run-time unpredictable execution flow. The statistic profile of distribution of Figure 4 shows also that the “90/50 Branch- Taken Rule” is not true for symbolic code, since most program branches (not belonging to the loop-backward class) are rather deterministic in their execution. Only a small percentage

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Traces</th>
<th>Basic Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Speedup vs. seq.</td>
<td>Average Length</td>
</tr>
<tr>
<td>con30</td>
<td>1.65</td>
<td>11.88</td>
</tr>
<tr>
<td>divide10</td>
<td>1.98</td>
<td>13.35</td>
</tr>
<tr>
<td>log10</td>
<td>1.81</td>
<td>12.95</td>
</tr>
<tr>
<td>mu</td>
<td>2.28</td>
<td>6.94</td>
</tr>
<tr>
<td>reverse</td>
<td>1.79</td>
<td>12.55</td>
</tr>
<tr>
<td>opt8</td>
<td>2.07</td>
<td>12.71</td>
</tr>
<tr>
<td>proof</td>
<td>2.20</td>
<td>14.64</td>
</tr>
<tr>
<td>quest</td>
<td>1.93</td>
<td>14.87</td>
</tr>
<tr>
<td>queen8</td>
<td>1.90</td>
<td>10.43</td>
</tr>
<tr>
<td>sendmore</td>
<td>3.18</td>
<td>8.83</td>
</tr>
<tr>
<td>serialele</td>
<td>2.68</td>
<td>11.11</td>
</tr>
<tr>
<td>tak</td>
<td>2.30</td>
<td>9.05</td>
</tr>
<tr>
<td>times10</td>
<td>2.08</td>
<td>13.35</td>
</tr>
<tr>
<td>zebra</td>
<td>2.27</td>
<td>10.08</td>
</tr>
<tr>
<td>Average</td>
<td>2.15</td>
<td>11.62</td>
</tr>
</tbody>
</table>

Table 1 - Speed-ups which can be obtained by applying Basic Block compaction or Trace Scheduling for shared memory architectures.
Table 2 - Probability of faulty prediction of branch targets for the considered benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Average Probability of Faulty Prediction of branch direction ($P_{fb}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>conc30</td>
<td>0.0292</td>
</tr>
<tr>
<td>crypt</td>
<td>0.0408</td>
</tr>
<tr>
<td>divide10</td>
<td>0.0935</td>
</tr>
<tr>
<td>log10</td>
<td>0.0354</td>
</tr>
<tr>
<td>mt</td>
<td>0.1215</td>
</tr>
<tr>
<td>reverse</td>
<td>0.0523</td>
</tr>
<tr>
<td>oops</td>
<td>0.1297</td>
</tr>
<tr>
<td>prover</td>
<td>0.0976</td>
</tr>
<tr>
<td>qsort</td>
<td>0.2376</td>
</tr>
<tr>
<td>queens_8</td>
<td>0.0973</td>
</tr>
<tr>
<td>query</td>
<td>0.1164</td>
</tr>
<tr>
<td>sendmore</td>
<td>0.0888</td>
</tr>
<tr>
<td>serialise</td>
<td>0.0896</td>
</tr>
<tr>
<td>tag</td>
<td>0.1994</td>
</tr>
<tr>
<td>times10</td>
<td>0.0935</td>
</tr>
<tr>
<td>zebra</td>
<td>0.1898</td>
</tr>
<tr>
<td>Average</td>
<td>0.1475</td>
</tr>
</tbody>
</table>

(Represented as the peak around 0.4) is really data-dependent and represents the fraction of branches which actually decides the semantic of the programs.

These considerations, which are not intuitive, show how a Trace Scheduling approach could be effective also when applied to symbolic computation. Disadvantages of a larger code size due to the introduction of compensation copies, are overcome by the advantage of a faster execution of the most frequently executed parts. Moreover, if we consider a hierarchical memory organization, only a small frequently executed part of code will be kept in cache, while other large parts of program will be stored (and compacted) in slower memory.

4.5. Estimated performance of VLIW architectures

Starting from the results of the previous sections which demonstrate the opportunity of applying global compaction techniques to a shared-memory Prolog abstract machine, we can design an architecture to support the considered computational model.

By using the described evaluation system, we can estimate the performance of different machine configurations, expressed in terms of machine cycles, in order to get technology-independent comparisons.

As described in section 3, we consider the class of parallel synchronous non-homogeneous architectures with dedicated hardware for supporting Prolog intrinsic features. In order to maintain capabilities of general-purpose computation, we limit specialized hardware structures to the minimum set which is necessary for an efficient management of tagged data.

The target architectures have the block structure shown in Figure 5, and can be composed of a variable number of basic units. Each unit supports simple arithmetic, logic and control operations, with no interpretation level between instruction and hardware, is completely exposed to the compiler, and can execute in the same cycle a memory access, a control operation, an ALU operation and a local data movement. The Instruction Set is register-oriented, with only direct and immediate addressing modes, and the only instructions which can access memory are explicit load and store operations.

Prolog dedicated features are added to optimize datapath structure and branch management.

- The data path supports tagged data and registers are organized into independent data fields.
- In order to reduce the overhead of redundant compare operations, the sequencer can branch directly on the tag fields. Also delayed branches, overlapping and multi-way jumps are allowed.

Fig. 5 - The structure of the architecture support used to evaluate the effectiveness of global compaction techniques.
Table 3 - Cycles and speed-up versus a sequential machine of different parallel architectural configurations. The architectures are assumed to be composed of the units shown in Fig.5.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>seq</th>
<th>BAM</th>
<th>1 unit</th>
<th>2 units</th>
<th>3 units</th>
<th>4 units</th>
<th>5 units</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>cycles</td>
<td>s.u.</td>
<td>cycles</td>
<td>s.u.</td>
<td>cycles</td>
<td>s.u.</td>
</tr>
<tr>
<td>conc30</td>
<td>798</td>
<td>577</td>
<td>1.38</td>
<td>513</td>
<td>1.56</td>
<td>512</td>
<td>1.56</td>
</tr>
<tr>
<td>divide10</td>
<td>1902</td>
<td>1173</td>
<td>1.62</td>
<td>1101</td>
<td>1.73</td>
<td>1016</td>
<td>1.87</td>
</tr>
<tr>
<td>log10</td>
<td>626</td>
<td>610</td>
<td>1.03</td>
<td>368</td>
<td>1.70</td>
<td>541</td>
<td>1.84</td>
</tr>
<tr>
<td>mu</td>
<td>49099</td>
<td>25929</td>
<td>1.89</td>
<td>38281</td>
<td>1.28</td>
<td>27172</td>
<td>1.81</td>
</tr>
<tr>
<td>reverse.</td>
<td>8925</td>
<td>6234</td>
<td>1.43</td>
<td>5542</td>
<td>1.61</td>
<td>5515</td>
<td>1.62</td>
</tr>
<tr>
<td>opt8</td>
<td>1241</td>
<td>762</td>
<td>1.63</td>
<td>705</td>
<td>1.76</td>
<td>636</td>
<td>1.95</td>
</tr>
<tr>
<td>priver</td>
<td>53791</td>
<td>29460</td>
<td>1.83</td>
<td>33425</td>
<td>1.61</td>
<td>28757</td>
<td>1.88</td>
</tr>
<tr>
<td>qsort</td>
<td>95964</td>
<td>6828</td>
<td>1.4</td>
<td>5244</td>
<td>1.82</td>
<td>5111</td>
<td>1.87</td>
</tr>
<tr>
<td>queen 8:</td>
<td>56924</td>
<td>36416</td>
<td>1.56</td>
<td>34338</td>
<td>1.66</td>
<td>31967</td>
<td>1.78</td>
</tr>
<tr>
<td>sendmore</td>
<td>1859889</td>
<td>1282921</td>
<td>1.45</td>
<td>1075486</td>
<td>1.73</td>
<td>845423</td>
<td>2.2</td>
</tr>
<tr>
<td>serialise</td>
<td>30080</td>
<td>15554</td>
<td>1.93</td>
<td>17847</td>
<td>1.69</td>
<td>14342</td>
<td>2.1</td>
</tr>
<tr>
<td>tak</td>
<td>1364190</td>
<td>940817</td>
<td>1.45</td>
<td>732671</td>
<td>1.86</td>
<td>653498</td>
<td>2.09</td>
</tr>
<tr>
<td>times10</td>
<td>1704</td>
<td>1047</td>
<td>1.63</td>
<td>958</td>
<td>1.78</td>
<td>872</td>
<td>1.95</td>
</tr>
<tr>
<td>zebra</td>
<td>5031109</td>
<td>2633032</td>
<td>1.91</td>
<td>2877146</td>
<td>1.75</td>
<td>2526207</td>
<td>1.99</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>1.58</td>
<td></td>
<td>1.68</td>
<td></td>
<td>1.89</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 6 - Performance comparison of different architectures (whose structure is described in Figure 5), a pure sequential implementation and the BAM processor itself.

The system architecture can be composed of an arbitrary number of units, depending on the required cost/performance ratio. A single Very Long Instruction Word is issued at every cycle with a unique control flow.

The instruction set of the architecture is similar to the BAM, but with a lower level of complexity. Since the micro-architecture is completely compiler-driven, BAM instructions that require sequences (e.g.: dereference, unification) are implemented via primitive operations.

In order to verify the validity of global compaction techniques, we need to measure the variation of the speed-up with an increase in the number of units of the architecture. As terms of comparison, we use both a pure sequential machine, and the BAM processor itself. On this purpose, we suppose that the duration of the operations of the architecture is equal to the BAM ones.

Some consideration can be drawn from the analysis of Table 3 and Figure 6, which show the results of our analysis:

- For most benchmarks, there is a speed-up saturation for a number of units larger than 3-4, as it was forecast by Amdahl's law.
- We can notice how the BAM shows a speed-up of about 1.6 with respect to a pure sequential implementation. This is very close to the limit of basic blocks (1.65, see Table 2) and is obtained by an instruction set which has some parallelism features, and by the adoption of RISC optimizations techniques (like delayed branches).
- In any case, the cost/performance gain with respect to the BAM is not impressive for architectures with more than one unit.
- However, in high-end applications where performance is the key parameter, a global compilation technique represent the only viable chance with a shared memory model.
5. A VLIW PROTOTYPE

5.1. Architecture features

A first result of the SYMBOL project is a VLIW incremental architecture and compiler, intended to work as a Hardware Accelerator attached to a host workstation. The architecture is designed on a VLSI processor, whose features conform to the description of a unit in section 4.5.

The architecture represents an experiment in building a real machine to test on the field the applicability of the described compilation techniques. The prototype will be used for research purposes and applied to control tasks in autonomous vehicle navigation problems.

In order to use completely automated layout tools for rapid prototyping, we have adopted a standard cell approach and we have been forced to reach a compromise between performance and cost issues. As a consequence, the implemented processors show some limits in comparison with the unit described in section 4.5:

- For scalability and testability reasons, a single processor is self-sufficient, and contains registers, an ALU and a sequencer. When more sequencers are used for multi-way branches, synchronization among them is achieved by means of an external logic network. In fact, since multiple branch instructions can be issued in the same cycle, a strategy has to be used to solve conflicts when more than one condition is true. So the compiler includes bits in the instructions to specify the priority of the branch operations. At each branch, a controller decides which processor Program Counter will be enabled in the next cycle.

- For pinout limitations, we had to organize the instruction in two formats, one for ALU operations and one for control operations. Then the compiler has to choose, and parallelism is somewhat reduced. However, memory accesses can be issued in both formats.

- For on-board memory cost considerations, accesses to code and data space have been organized in a three-cycle pipeline. This does not compromise the peak rate of one access per cycle, but causes two-cycles delayed branches and a longer completion time of data memory operations.

5.2. The VLSI Processor

Machine instructions are horizontal, 64 bits wide and organized into two formats, one for direct and one for immediate addressing.

- Direct address format allows a memory access, an ALU operation and a register movement.
- Immediate address format allows a control operation (or immediate operand movement) and a memory access.

The datapath (Figure 7) is composed of:

- a multi-port Register Bank (16 registers),
- an Arithmetic-Logic Unit,
- a Sequencing unit,
- a Data Memory interface.

The Register bank is composed of 16 32-bits registers organized in separate and independently addressable fields:

- value (28 bits), used to store addresses or constants
- tag (3 bits), used to store data specifications (list, structure, atom, variables) as well as arithmetic flags deriving from compare operations
- cdr (1 bit), used for list concatenation and for some status flags, originally introduced for compatibility with the WAM.

Fig. 7 - The micro-architecture of the prototype processor

Unlike common Prolog processors, there are no reserved registers (apart from the Program Counter) for environment variables. In this way the code generator is free to decide where to store a variable, to reduce the traffic on the buses connecting the processors. The processor has no dedicated status word, and flags resulting from ALU operations can be stored (only when necessary) in the tag field of any register.

As in other RISC machines, the sequencer has a reduced complexity, because of the simple control instructions supported that actually are only branch-on-condition and jump (direct and indirect). A little hardware support has been added for pipelined jumps management: when a jump is executed in any processor, sequencers of all processors must be disabled in the following two cycles, in order to preserve program flow correctness.

Data Memory is interfaced through non-multiplexed data and address buses, which are also used for processor-to-processor register movements. Addresses and the different fields of data can come from different processors with no constraints. Memory operations are organized in a three-stages pipeline, with the assumption of an external memory (or cache) with a single cycle access time. MAR and MDR registers are external to the processor, and transparent to the compiler.

The control part is just composed of decoders for the instructions fields, which are fixed for each format and directly control functional units and bank ports of the architecture. Code Memory access is organized in a three-cycle pipeline, also considering the decoding phase (which is very simple indeed). An external Program Counter and Instruction Register are included for the pipeline.

The RISC processor has been designed by using a hierarchical standard cell environment. The chip obtained has an area of 12.5 mm by 13.5 mm and is composed of about 100,000 transistors (Figure 8) [DeGloria91].

231
The measured operating frequency is 30 MHz.

5.3. Performance Measurements

Following [Hennessy90], performance of an architecture can be measured only by taking into account all of the following factors:
- Cycles Per Instructions (CPI). In our case CPI = 1.
- Clock Rate. In our case Clock Rate = 30 MHz
- Instruction Count.

For this reason, instead of giving performance in terms of MLIPS (Millions of logical inferences per second), we prefer to show absolute execution times. However, for the NREVERSE benchmark, which is commonly used as a reference term for measuring logical inferences, we obtain a peak performance of 2.1 MLIPS.

Table 4 shows the (simulated) execution times of a three-processor architecture for the benchmarks (expressed in milliseconds) in comparison with other Prolog machines. We can see that absolute performance of a three-processor architecture reaches only 83% of BAM performance. However, this fact does not affect the validity of the approach, since we must consider that:
- we have implemented an architecture which, although composed of three processors, is characterized by a remarkable design simplicity and has been realized with completely automated layout tools;
- we have used a slow technology (2 microns), and operation durations are consequently longer than the BAM processor.

In order to demonstrate the effectiveness of the Global Compaction technique, we can consider the speed-up of the architecture relative to a sequential implementation which obeys the same operation duration hypotheses (Table 5). We notice how a Trace Scheduling compilation succeeds in reaching a level of speedup (1.9) which is slightly higher than the BAM (1.5).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Quintus</th>
<th>VLSI-PLM</th>
<th>KCM</th>
<th>BAM</th>
<th>Symbol-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>divide10</td>
<td>0.41</td>
<td>0.38</td>
<td>0.091</td>
<td>0.0387</td>
<td>0.0423</td>
</tr>
<tr>
<td>log10</td>
<td>0.15</td>
<td>0.109</td>
<td>0.039</td>
<td>0.0201</td>
<td>0.0146</td>
</tr>
<tr>
<td>mnist</td>
<td>12.407</td>
<td>4.644</td>
<td>0.8557</td>
<td>1.2913</td>
<td></td>
</tr>
<tr>
<td>reverse</td>
<td>1.62</td>
<td>2.10</td>
<td>0.65</td>
<td>0.2057</td>
<td>0.2401</td>
</tr>
<tr>
<td>op8</td>
<td>0.24</td>
<td>0.214</td>
<td>0.059</td>
<td>0.0251</td>
<td>0.0274</td>
</tr>
<tr>
<td>prover</td>
<td>8.67</td>
<td>6.83</td>
<td>-</td>
<td>0.9722</td>
<td>1.2995</td>
</tr>
<tr>
<td>sort</td>
<td>4.82</td>
<td>4.24</td>
<td>1.32</td>
<td>0.2253</td>
<td>0.2192</td>
</tr>
<tr>
<td>queens_8</td>
<td>21.20</td>
<td>28.80</td>
<td>1.205</td>
<td>1.2017</td>
<td>1.549</td>
</tr>
<tr>
<td>sendmore</td>
<td>490.00</td>
<td>-</td>
<td>-</td>
<td>42.3364</td>
<td>44.0939</td>
</tr>
<tr>
<td>serialize</td>
<td>3.10</td>
<td>2.47</td>
<td>1.22</td>
<td>0.5133</td>
<td>0.6556</td>
</tr>
<tr>
<td>tak</td>
<td>1120.00</td>
<td>940.00</td>
<td>-</td>
<td>31.047</td>
<td>32.067</td>
</tr>
<tr>
<td>times10</td>
<td>0.345</td>
<td>0.2470</td>
<td>0.082</td>
<td>0.0346</td>
<td>0.0363</td>
</tr>
<tr>
<td>zebra</td>
<td>425.00</td>
<td>-</td>
<td>-</td>
<td>86.890</td>
<td>119.184</td>
</tr>
<tr>
<td>average speed-up</td>
<td>(19.51)</td>
<td>(10.47)</td>
<td>(2.42)</td>
<td>(0.83)</td>
<td>(1.89)</td>
</tr>
</tbody>
</table>

Table 4 - Performance comparison of some Prolog implementations. Values are expressed in milliseconds. The last rows shows the speed-up relative to the SYMBOL-3 (three-processors) VLIW architecture. The values are computed without taking into account the cycle time difference, since it may be influenced by the adopted architectural model.

Fig. 8 - A photograph of the layout of the prototype processor
6. CONCLUSIONS

In this paper we have presented results of a code analysis which show the effectiveness of a Global Compilation approach to the BAM model of Prolog.

The basic consideration which emerges from the analysis is that Prolog shared-memory models have almost reached their limits with the BAM architecture.

In this context, further improvements can come only from technology (designing faster processors), or architecture (adopting dynamic scheduling), and only marginally from a static exploitation of instruction-level parallelism. In any case, we can't overcome Amdahl's limit of speedup (about 3) with a shared memory model.

In our opinion, the only solution to break the barrier with a compiler-driven parallel approach is in the development of distributed memory models for Prolog. There, a global compaction algorithm can be really effective and reach order of magnitude of improvements, but it is still an open field for future research.

ACKNOWLEDGEMENT

We would like to thank the Aquarius research group at University of California, Berkeley for key contributions to this research. In particular, thanks to Peter Van Roy for the BAM compiler and to Bruce Holmer for the BAM simulator.

REFERENCES

