EXECUTING CONTEXTUAL LOGIC PROGRAMMING ON A DEDICATED VLSI COPROCESSOR

Pierluigi Cierra(*), Evelina Lamma(**), Paola Mello(***),
Antonio Natali(**), Gianluca Piccinini(*), Maurizio Zamboni(*)

(*) Dip. di Elettronica
Politecnico di Torino
Corso Duca degli Abruzzi, 24
10129 Torino
email: vlsl@itopoli.bitnet

(**) DEIS - Universita' di Bologna
Viale Risorgimento, 2
40136 Bologna
email: boari@bodeis.cineca.it

Abstract
In this work we address the effective implementation of a logic language based on contextual logic programming that subsumes standard Prolog and efficiently supports and integrates different structured logic languages. The implementation is based on an extension of the abstract machine developed by D.H. Warren (WAM), and is obtained by adding a specialized coprocessor, based on a microprogrammed VLSI architecture, to a standard CPU. In the paper we will describe the extended WAM, the compilation-based environment supporting the general framework, and the specialized microprocessor architecture.

Introduction
A crucial research topic of logic programming is how to introduce structuring concepts in it such as modules and blocks ([9], [15]), inheritance between separate logic theories ([8], [19]), and viewpoints ([7], [12]). In [11] and [13] an extended logic programming language that subsumes and integrates some of the most well-known proposals for structuring logic programs is designed and its declarative and operational semantics is formalized.

This language (hereinafter called EC-Prlog) is based on contextual logic programming ([14]) suitably extended to achieve the required generality. In [11] the implementation of EC-Prlog is defined on the basis of an extended Warren Abstract Machine ([18]) (hereinafter called S-WAM). While in [11] the focus is on the S-WAM design, in this paper we describe the real, efficient implementation of the general framework in a compilation-based environment.

Starting from the intermediate (WAM/S-WAM) code, three implementation approaches are possible:
1. Translate the intermediate code into the native code of a target, general-purpose machine;
2. Execute the intermediate code by using software emulators;
3. Execute the intermediate code on a dedicated (VLSI) machine.

In the attempt of improving performance, several research projects ([4], [16], [2]) propose fully dedicated Prolog machines, although it is not clear at the state of the art if this implementation technique offers a significant improvement with respect to the first approach. In this work, we follow the third approach for implementing EC-Prlog. The implementation here described is based on a special-purpose VLSI microcircuit architecture dedicated to the execution of S-WAM instructions. This dedicated architecture (hereinafter called S-PROXIMA) is obtained as a rather natural extension of the PROXIMA Prolog machine ([2]).

Structured logic programs based on blocks, modules, inheritance and hypothetical reasoning can be implemented and integrated on this efficient architecture.

EC-Prlog: an overview
In this section, we briefly present EC-Prlog. A deeper discussion together

Example 2.1
Let us consider the following program:

unit(u1):
  unit(u2):
  unit(u3):
    a:-b.
    b.
    c:-a.
  c:-a.
  b.
  a:-b.

In the context [u3,u2,u1], a proof for the goal c is performed by virtually considering the following set of clauses:

Example 2.2
Given the following program:

Extension and overriding for predicate definitions
In logic programming non-determinism is present since each procedure $p/n$ may correspond to different clauses. In this setting two forms of non-determinism are provided. A procedure, in fact, may have multiple definitions not only in the same unit (intra-unit non-determinism), but also in different units of the same context (inter-unit non-determinism). Two different policies can be adopted with reference to predicate definitions in the context:

1. The most recent predicate definition overrides the previous ones for the same predicate. Only intra-unit non-determinism may be present.
2. The most recent predicate definition extends the previous ones for the same predicate. Both intra- and inter-unit non-determinism may be present.

In EC-Prlog, the default policy is predicate overriding. To obtain the predicate extension policy for predicate $p/n$, the following declaration must be inserted:

\[ \text{Extends}(p/n) \]

If this declaration is present in a unit $U$, not only the definition of $p/n$ in $U$, but also those in units before $U$ in the current context will be taken into account. Moreover, to support information hiding, a predicate definition $p$ is exported from a unit $U$ (i.e. visible outside $U$) only if the declaration

\[ \text{Visible}(p/n) \]

is defined in $U$. In the rest of the paper, for the sake of simplicity, all predicates are supposed to be visible.

Example 2.2
Given the following program:
failure
b(2).

a(X):-be).

while the context [u3,u2,ul] the following set:
b(1).
a(X):-b(X).

b(1).

Conservative and Evolving Policies for Binding Predicate Calls

In order to evaluate a predicate call in a context we have to find, in that context, the appropriate set of definitions, i.e. the binding for predicate calls. Two different policies of binding - referred to as conservative and evolving - can be adopted in EC-Prolog.

Let us suppose that C=[ul,...,u2,u1] is the current context.

If an evolving policy is adopted, the predicate definition for a call occurring in unit ul is given by the clauses of the whole context C. We will refer to C as the global context (GC).

If a conservative policy is adopted, the predicate definition for a call occurring in unit ul is given by the clauses of the sub-context [u1,...,u2]. We will refer to this sub-context as the partial context (PC).

EC-Prolog supports both policies and adopts the conservative one as default. Evolving policy calls are prefixed by the symbol #. In order to support both the policies, two different contexts (i.e. the global and the partial one) have to be maintained by the run-time support of the language.

In this setting a top-down derivation (see [1]) is given in terms of sequences of formulae of kind: GC PC F, where GC is the current global context, PC the current partial one and F a goal formula.

Example 2.3

Let us consider the following program P:

unit(u1): unit(u2): unit(u3):
b(2).
a(X):-b(X).

Since a definition for b is sought in the partial context [u1], the derivation fails.

Adopting an evolving policy for b, instead, by prefixing its call in the unit u1 by #, leads to the following successful top-down derivation:

[u3,u2,u1] [u3,u2,u1] # c
[u3,u2,u1] [u3,u2,u1] a
[u3,u2,u1] [u1] # b
success

Building the Context

A context can be built by using the context extension operator $latex \texttt{extends(b/l}, \texttt{u1/l,} \texttt{u2/l)}$. Since two contexts are taken into account in EC-Prolog, two different extension operators ($latex \texttt{extends(b/l}, \texttt{u1/l,} \texttt{u2/l)}$ and $latex \texttt{extends(b/l}, \texttt{u2/l,} \texttt{u1/l)}$) are provided. The goal $latex \texttt{a(X):b(X).}$ extends the current partial context with unit u1, and then makes GC equal to PC. By converse, the goal $latex \texttt{a(X):b(X).}$ extends the global context with unit u1, and then makes PC equal to GC.

Moreover, in EC-Prolog, a context can be statically associated with a unit when it is defined, or dynamically built by using the context extension operator. In the following, we will consider only the latter, more flexible and dynamic case.

Implementation on a Dedicated Coprocessor

In this section, we present the real implementation of EC-Prolog with particular reference to the compilation-based environment and the dedicated VLSI coprocessor. The implementation is based on an extended Warren Abstract Machine (S-WAM) which is, in practice, an optimized extension of the one presented in [11].

In the S-WAM a new stack, representing the current global and partial contexts, has been added to WAM, along with new instructions to expand/contract it. Moreover, the structure of both the choice point and the environment of the WAM have been expanded to consistently handle new registers and some optimizations have been also considered, for limiting the overhead in the case of execution of standard Prolog programs.

An emulation environment has been used to develop the model, select and evaluate different architectural solutions, and produce the microcode of the dedicated coprocessor.

The complete development environment is shown in figure 1.

The compiler transforms EC-Prolog programs into S-WAM instructions. The output of the compiler is a text representation of the resulting S-WAM instructions. The byte code generator translates S-WAM instructions into the instructions directly executable by the processor, using separate symbol tables for constants and units. In particular, it produces a global table where, for each unit, the address of the corresponding compiled code in the code area is reported.

The S-WAM emulator executes the programs and saves the traces of the operations on a history file. Moreover, a low-level emulator (level zero emulator) is provided to execute the basic operations of the architectural model. Filtering programs are provided to calibrate the model and choose the best set of operators/registers to be implemented in hardware. The most important features to be optimized are the memory structure usage and the internal parallelism of the microarchitecture achievable. The data obtained let the designer tune the model optimally.

The Architectural Prolog Evaluator (APE), written in Prolog, analyzes the emulation traces to perform architectural comparisons of different hardware solutions. The tool analyzes the traces comparing each operation with an input data-base representing the target architecture. After generating the microcode (Control Unit Code Generator), a microcode machine emulator is finally used to test the microarchitecture and to optimize the microcode. The optimization mainly concerns jump prediction and microroutine efficiency.

Figure 1: The development environment
Compilation

The compiler has been obtained by extending a standard Prolog compiler [17], written in Prolog, which translates Prolog programs into WAM instructions.

The extensions of the compiler deal with units, context extension operators, bindings for predicate calls with respect to the global or partial context, and inter-unit non-determinism.

The compiler classifies a predicate call p occurring in a unit U as:
1. local if a local predicate definition for p exists in U;
2. eager if no local predicate definition for p exists in U;
3. lazy if p is prefixed by the # operator.

Local predicate calls of a unit U can be directly bound with respect to clauses of U at compilation time, as it happens in the standard WAM. By converse, bindings for eager and lazy predicate calls cannot be statically solved, since they depend on the context on which U is allocated/nested.

Bindings for eager calls can be solved, in fact, when the context is extended with U. Such bindings are recorded in a specific, run-time area associated with the unit (called instance environment) to be used for further calls.

Bindings for lazy calls, instead, are to be solved newly each time the call occurs. This explains their name.

The code of a unit U (see figure 2) consists of the compiled code of the procedures defined in U. Moreover, some information is associated with U code:
1. a table for visible procedures defined in U;
2. a local table, ET, maintaining each predicate name defined $extends in U or corresponding to an eager call occurring in U.

Local calls are compiled into standard WAM instructions (call plh,m or execute plh instructions), while for eager and lazy calls new, specific instructions are added (respectively call-lazy plh.m or execute-lazy plh/n and call_lazyp.lh.m or execute_lazy plh).

For each extending procedure plh, try-me_else instructions are used to allocate a choice point even if plh is deterministic.

The code for the last clause of plh is followed by the new instruction trust_extends plh, which forces a dynamic search for an alternative definition of plh along the current context (inter-unit non-determinism).

Each external goal formula u=g or u=G is translated into the following instructions:
allocate_c_ctx u (allocate_l_ctx u)
call_lazy p
deallocate_c_ctx

Figure 2: A compilation example

The Run-time Structure

A new memory area, called the context stack, is added to the WAM to support units and their combination into contexts.

This stack represents the binding environment for conservative and evolving policies. The context stack grows whenever an extension U=>G (or U>>G) occurs and shrinks when G is deterministically solved or definitely fails. Three new registers refer to this stack: PC, representing the partial context; GC, the global one; and IE_top, the top of the context stack.

Each object on the context stack is an instance environment associated with a unit U and allocated on the context stack whenever U is involved in a context extension. The context stack and the instance environment structure is sketched in figure 3.

Figure 3: The context stack of the S-WAM

An instance environment, IE, for U consists of a number of cells, where the bindings for eager predicate calls occurring in U are recorded. These bindings depend on the context on which U is allocated and are actually solved the first time the call is performed for dynamic units.

In particular, each instance of a unit U shares the same code and has a private set of references for eager goals in U.

In particular, IE consists of:
1. A number of cells, statically determined by the compiler, representing the number of eager calls and extending procedures occurring in the unit code. The position i of a cell in the instance environment of U corresponds to the i-th predicate name plh in the local table, ET, of U.
2. A slot where the value of PC is saved;
3. A slot where the value of GC is saved;
4. A reference (unit_ref) to the code of a in the code area;
5. A slot (chain) maintaining a reference to the current context (PC or GC) on which a is nested. Chain slots maintain the links between instance environments on the context stack;
6. A slot where the value of IE_top is saved.

Let us notice that values of the registers PC, GC and IE_top are modified whenever the context is extended. PC may also vary when executing an eager or lazy call or searching for an alternative clause of an extending predicate. Since the program counter can now be seen as the pair <PC, <program pointer, partial context>>, also for PC a continuation register (CPC), saved in the environment, is introduced.

To consistently handle backtracking, IE_top, CPC, PC and GC are saved in the choice point. To reduce the overhead introduced in the case of standard Prolog programs, the choice point is split into two data structures. The first one, allocated on the local stack, corresponds to the original data structure of the WAM, in which a slot has been added for saving IE_top.
The main problem that arises when language primitives are executed by
the processor is shown in Figure 4. Three main functional units can be identified in S-PROXIMA:
- control unit;
- execution unit.

The instruction set has been coded to optimize the bandwidth and the code
memory space requirements in a variable length format.

Figure 4: S-PROXIMA block diagram

The computational requirements of new programming paradigms can be satisfied designing physical architectures based on specific VLSI (Very Large Scale Integration) devices [6], [5]. For this reason, the execution of structured logic programming here presented is based on a dedicated microcoded coprocessor (called S-PROXIMA, i.e. Structured PROlog xEecution MAchine) which directly implements the S-WAM.

S-PROXIMA has been derived from PROXIMA architecture [2], which directly implements the basic WAM in hardware, by enhancing its instruction set for contexts handling. The design methodology followed to redesign the processor is based on the mapping of the abstract machine (S-WAM) on a physical one.

The main problem that arises when language primitives are executed by a dedicated architecture, is related to the definition of the abstraction level of the target code. In S-PROXIMA, the S-WAM instruction set has been identified as the most effective level of target code also considering the complexity of the S-WAM instructions and the definition of a fully specific execution unit allow a high throughput to be reached. The resulting microcode is 128 bits wide and 900 lines long.
First Results and Conclusions

This work shows the real implementation of an extended logic language based on contextual logic programming which subsumes standard Prolog and embeds different structuring mechanisms. The implementation described is based on an extension of the Warren Abstract Machine (S-WAM), and is obtained by adding a specialized VLSI coprocessor (S-PROXIMA), based on a microprogrammed architecture, to a standard CPU.

The design of S-PROXIMA has been supported by a set of measurements performed on different benchmark marks to detect the best allocation of the new registers required by the context instructions, without deep modification of the basic structure of PROXIMA.

Performance results, reported in [3], show that when executing Prolog programs on S-PROXIMA the decrease of performance is limited but significant (3-9%). The causes of the computational overhead for standard Prolog execution are mainly due to internal activities (register transfers, increments, decrements, tests, etc.) and not to memory accesses. This overhead could be reduced by further architectural optimizations that we plan to investigate in the future.

A significative increase of memory accesses is present when dealing with contexts and it is mainly due to the management of the new memory structures such as the context stack and the unit table. The number of the internal transfers increases about 50% because of the new registers used in the context handling. This is partially due to the non complete optimization of the S-PROXIMA architecture for the new S-WAM instructions.

With reference to efficiency increasing, we plan to follow two complementary approaches. The first approach concerns the application of partial evaluation techniques ([10]) to EC-Prolog programs. The second approach involves a revision of the S-PROXIMA architecture on the basis of measurements for more significant programs, in order to optimize S-WAM instruction execution and to obtain higher performance also when contexts are deeply used.

Acknowledgements

This work has been partially supported by the "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo", under grant n. 890004269, and by the "Progetto Finalizzato Materiali e Dispositivi Elemnici all0 Informatici e Calcolo Parallelo", under grant n. 890004269, and by the "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo", under grant n. 890004269, and by the "Progetto Finalizzato Materiali e Dispositivi Elemnici all0 Informatici e Calcolo Parallelo", under grant n. 890004269, and by the "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo", under grant n. 890004269, and by the "Progetto Finalizzato Materiali e Dispositivi Elemnici all0 Informatici e Calcolo Parallelo", under grant n. 890004269, and by the "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo", under grant n. 890004269, and by the "Progetto Finalizzato Materiali e Dispositivi Elemnici all0 Informatici e Calcolo Parallelo", under grant n. 890004269, and by the "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo", under grant n. 890004269.

References