A High Speed Prolog Implementation on a VLIW Processor*

W. Schenk†
Institut für Informatik und Praktische Mathematik der Universität Kiel
Olshausenstr. 40-60, D-2300 Kiel, W. Germany

Abstract: *A microprogrammable target computer allows implementing a virtual machine efficiently. When implementing a compiler based high level language, the semantic level of the machine language has to be fixed. If the machine realizes a suitable virtual machine, the compiler writers task is simplified. This may be payed by the expense of implementing the virtual machine. The new retargetable compiler of the Mimola Software System (MSS) simplifies and speeds up the implementation process of a desirable machine language on a microprogrammable target computer. The efficiency of this approach is shown by an implementation of the logic programming language Prolog on the VLIW processor SAMP. The Warren Abstract Machine instruction set (W-Code) was choosen as the machine language of SAMP. The usage of the new retargetable compiler of the MSS for the development of a microcoded W-Code interpreter is described in this paper. The implementation has been tested and evaluated by feeding a simulator with the microcoded interpreter and the W-Code of the Warren Benchmark Set. The results indicate the performance of 44.5 KLIPS.*

1 Introduction

Todays high level language compilers are at least conceptually divided in a language dependent part and a machine dependent part. The language dependent part handles syntactic analysis, performs semantic actions, and generates an intermediate program representation. The knowledge about the target machine language as used for codegeneration is encapsulated in the machine dependent part. The codegenerator processes the intermediate program while emitting target machine code. If the intermediate program representation is designed as a machine language of a virtual machine then the implementation of that machine replaces most of the compilers machine dependent part. There are basically three ways of implementing a virtual machine.

- The construction of the virtual machine.
- The compilation of the virtual machine language into target machine code.
- The software emulation of the virtual machine.

In developing a microcoded interpreter for a virtual machine language we followed the third approach.

The most widely spread implementation techniques for Prolog involve compiling the Prolog program into an intermediate form (W-Code) referred to as the Warren Abstract Machine (WAM) instruction set. Several authors have investigated various special purpose hardware structures (Prolog machines) that support the W-Code as their machine language. The development of a microcoded interpreter for W-Code execution on the VLIW processor SAMP [Now87] which was choosen as the implementation vehicle for the WAM is the subject of this paper. The Prolog compiler that generates the W-Code, and the complete source code of the W-Code interpreter are described in detail elsewhere [Sch88]. The new retargetable compiler of the Mimola

*Microprocessing and Microprogramming Vol. 27, Nos. 1-5 (1989) pp. 601-606

†now at: University Dortmund, FB Informatik, LS XII, Baroper Str. 301, GB IV, D-4600 Dortmund 50, Tel.: (0231) 755 5101,E-mail: schenk@ls12.informatik.uni-dortmund.de

ing the resulting microcode, the W-Code of a Prolog program, and a hardware description of SAMP into the simulator of the MSS. A performance of 44.5 KLIPS on the Warren Benchmark Set is achieved. In what follows we describe architecture and tasks of the WAM, hardware structure and important features of SAMP, and the process of generating microcode. Finally, simulation results are presented which allows the performance comparison of SAMP with other processors.

2 Warren Abstract Machine

The WAM was originally introduced by D.H.D. Warren [War83]. The following brief explanation covers some implementation issues.

The WAM is an environment–stacking architecture. The stack consists of two kinds of variable–length frames (mixed stack). There are environments and choice points. An environment holds value cells for the permanent variables of a clause together with bookkeeping information comprising a continuation. A continuation consists of a pointer to the W-Code of the actual goal to be executed next and its associated environment. A choice point contains the state of the computation, at which the WAM continues on backtracking.

Since the type of a Prolog variable is not known at compilation time, its value is represented by a tagged item. A tagged item consists of a tag and a value field. The tag indicates the type of the value. Thus, checking an item's tag is the most frequent operation of the WAM. In the present implementation an unbound variable is represented by an item with an unbound tag and a self-reference in the value field. The value field of a bound variable either contains the value itself or a reference tag together with a pointer to the value. From this the need for dereferencing a variables value arises and dereferencing becomes another important task of the WAM. Figure 1 shows the dereferencing loop.

In Prolog a variable receives a value only by unification and there is no way to change it. Only variable bindings, which are established since the creation of the actual choice point, are reset on backtracking. The trail is a stack where the WAM remembers those bindings. Whenever a variable which will not be discarded by the next event of backtracking is bound it is trailed, i.e. its address test.

The value of a Prolog variable does not necessarly fit into a single variable cell. In this case a variable is represented by a pointer into the heap of the WAM. The heap holds the compound terms of the computation and is managed in a last-in-first-out mode. A new instance of a term is constructed at the top of the heap. While backtracking the heap shrinks to the size it had when the actual choice point was created.

Procedure invocation is done by copying the actual arguments into dedicated argument registers and passing a continuation to the callee. If the call is nondeterminate – i.e. the indexing instructions cannot narrow down the number of possible matching clauses of the called procedure to one – a choice point is created and loaded with the argument registers and bookkeeping information for backtracking. Specialized instructions for the head of a clause attempt unification against the actual arguments. If unification fails, backtracking takes place and the state of the WAM is recovered from the most recent choice point on the stack. The state of the WAM consists of the following entries:

P	Current Instruction Pointer
CP	Continuation Pointer
H	Heap Pointer
HB	Heap Backtrack Pointer
E	Environment Pointer
B	Current Choice Point
TR	Trail Pointer
cutp	Cut Pointer
A[1], A[2], ...	Argument Registers

In order to get an useful implementation, instructions for the cut and some build-in predicates are added to the WAM. The extended instruction set has 74 instructions, which are optimized with respect to SAMP.

3 SAMP

SAMP (self-timed, asynchronous, microprogrammed, parallel) was designed by L. Nowak [Now86] as a 16 bit general purpose computer. SAMP is 3.4 times faster than a SIEMENS 7760 and 7.2 times faster than a VAX-11/750 in performing various Pascal-like programs. From this the question arises whether SAMP is well suited only in performing programs derived from imperative styled high level programming languages, or it captures a wider range of applications.

hardware in a powerful way. A microinstruction has 141 bit and is directly interpreted by the hardware. The writeable control store contains up to 4096 microinstructions. The horizontal microinstruction format facilitates the implementation of flexible data paths. Four arithmetic logic units (ALUs) allow the parallel evaluation of multiple expressions. There are two comparators and a 1 bit ALU that form a logic network which generates a condition used by conditional jumps or conditional load operations. Furthermore there are two multiport memories that support multiple assignments in one microinstruction. E. Tick [Tic85] argues that memory bandwidth is the key for high ended Prolog machines. The main memory of SAMP achieves a memory bandwidth of 41 MB/sec. It has two writeable and two read-only ports that serve up to four independent accesses. The other multiport memory, the register file, has two readable ports and two ports for read/write access with a common address.

It is up to the microcode generator to detect simultaneously executable operations and to embed them into a microinstruction. The number of those operations is extended by means of guarded assignments. Guarded assignments are supported by SAMPs conditional load operations. Whether the load operation of a memory port is actually carried out or inhibited depends on the truthvalue of the condition. Since all storages of SAMP are able to conditionally load a value there is much source of parallelization per guarded assignments. The new retargetable compiler [Now87a] of the MSS takes this feature into account.

If timing could be ignored at the microprogram level, the generation of microcode became less difficult. SAMP is in fact a self-timed system. Timing is a matter of hardware in that the data flows asynchronously through self-timed elements. A signal is attached to each data connection, indicating its validity. The memories, ALUs, and multiplexers as well as the other function boxes are designed as self-timed elements. They perform a particular operation, when its precondition becomes true. That is when all of the necessary input values are available. The self-timed element signals the completion of the operation after its delay. There is no global clock signal needed for synchronization issues. The sequencing of the microinstructions is controlled by a global reset signal. It starts the next microinstruction as soon as all the operations of the current microinstruction and the data values it manipulates.

As an example consider a microinstruction which specifies a conditional load operation. Assume that the generation of the guarding condition is fast and the evaluation of the address and data is a complex task. If the condition yields true the microcycle time is determined by the time needed to compute the address and data plus the delay of the load operation. On the other hand the instruction is completed as soon as the condition becomes false. In this case no address or data is needed because the load operation is inhibited.

4 Microcodegeneration

For the generation of the microcoded W-Code interpreter the hardware description of SAMP and the algorithm for W-Code execution are specified in the Pascal-like language MIMOLA (machine independent microprogramming language), the input language of the MSS. The new retargetable compiler of the MSS yields microcode of high quality. The compiler does parallelization of the program, allocation of operations, and microcode compaction in different phases.

The retargetability of the compiler is based on a textual hardware description written in MIMOLA. The hardware structure is specified as a set of modules and a list of their interconnections. For a module, the interface and the operations it implements are described. The operation delays are given for simulation purposes only.

The W-Code interpreter is easily encoded as a MIMOLA program. An item is stored in two consecutive memory locations with the tag field at the lower address. This is specified by the type definition for items:

```
TYPE Item = RECORD tag,val : word END;
```

The storarge areas of the WAM are mapped into SAMPs main memory SM by means of array declarations:

```
Code  : ARRAY [0..CMax] OF word;
Heap  : ARRAY [0..HMax] OF Item;
 ...
Trail : ARRAY [0..TMax] OF word;
```

To increase efficiency the address computation facilities for arrays are inhibited and no range

Heap[S] is located at addresses *S* and *S+1* in the main memory.

Figure 1 shows the code for the dereferencing operation of the WAM. The loop follows a chain of reference items until an non-reference value is encountered. The result is left in the global variable *tmp*. The value field of a reference item is

```
VAR tmp : Item AT SR[0];

INLINE PROCEDURE deref(tag,val:word);
BEGIN
 tmp.tag:=tag; tmp.val:=val;
 WHILE IsReference(tmp.tag) DO
 BEGIN
  tmp.tag:=Mem[tmp.val].tag;
  tmp.val:=Mem[tmp.val].val
 END
END;
```

Figure 1: Dereferencing an Item

a pointer into the stack or into the heap of the WAM. In either case it is accessed via the array *Mem*. The inline function *IsReference* tests by selecting a single bit from the tag whether or not an item is a reference.

Applying transformation rules the MIMOLA source code is mapped into RT behaviour. The rules replace variables with memory accesses and they transform procedure calls into parameter passing operations. A call statement for an inline procedure is replaced with the procedures body, where the formal parameters are textually substituted by their actual values (call by name). Compound statements are expanded to assignments and IF-statements.

According to the flow of control the RT-program is split into basic blocks. A parallelization is done by dividing a basic block into a sequence of parallel executable blocks. An IF-statement may be implemented with the dependent then- or else-part as the target of a conditional jump or by a sequence of blocks consisting of parallel executable guarded assignments. The latter may extend the scope of a basic block and thereby reduce the number of microinstructions required. The conditional load operations of the dereferencing loop generated by the call *deref(Heap[S].tag,Heap[S].val)* are shown in figure 2. They are part of the expanded body of the inline procedure.

```
IF SR[0] "SELECTBIT" 15 THEN SR[1]:=SM[SR[1]+1];
RP:=IF SR[0] "SELECTBIT" 15
  THEN Line4711_deref
  ELSE "INCR" RP;
PAREND;
```

Figure 2: conditional load operations for deref

The item *tmp* is located at addresses 0 and 1 in the register file SR.

An IF-statement cannot be transformed into guarded assignments if a sequence is required by the definitions and uses of a value belonging to a dependent part of the statement. Whenever an IF-statement may be compiled as conditional jump or conditional load operation both alternatives are feeded into the allocation phase. If the compiler fails in allocating one of these it retains the other.

A remarkable feature of the allocation phase is the treatment of versions of the instruction that implement a statement. The versions are constructed from versions of the operations that make up the statement. For the dereferencing loop the compiler allocates an instruction for the conditional jump and one instruction for the loop body whereas the conditional load operations are compiled into a single instruction. The compiler considers e.g. 76, 53 and 14 versions for the first, second and third statement of figure 2 respectively, which merge to 214 versions of the parallel block. All versions are considered by the microcode compaction algorithm. It merges as many assignments as possible into a single microinstruction. If there still remain different versions, the fastest one is selected.

The W-Code interpreter has 443 parallel blocks with 1,074 statements. The compiler considers 39,922 versions of the statements. The resulting microprogram has 555 microinstructions.

5 Simulation and Performance Measurement

The performance was measured using the Warren Benchmark Set. This consists of Prolog programs by which various performance aspects such as access to logical variables, unification of structures, list handling, indexing of clauses, backtracking, arithmetic, etc. are tested. A Prolog compiler generates optimized W-Code from the Prolog

loaded into the main memory.

The event driven simulator of the MSS considers the self-timed sequencing and the delay of the modules operations. This gives the reason for high confidence in the resulting timing informations (see table 1). The simulator computes the propagation of bit vectors along the connections and the function boxes of the hardware.

Benchmark	Runtime [μs]	μ-instr. executed	LIs
nrev30	7,631	21,258	496
qsort50	10,650	29,502	601
palin25	7,264	17,893	322
times10	658	1,789	19
divide10	784	2,148	19
log10	267	749	11
ops8	424	1,158	15
query	37,136	105,801	2107
Total	64,814	180,298	3590

Table 1: Simulator Results on the Warren Benchmark Set

The benchmarks are executed in 65 ms with 180,298 microinstructions simulated. Therefore the cycle time is 0.36 μs. The performance is expressed in terms of KLIPS (kilo logical inferences per second), with the cut not being counted as a logical inference. On the Warren Benchmark Set 44.5 KLIPS were achieved on the average.

Benchmark	Performance [KLIPS]			
	SAMP	DEC	NCR	VAX
nrev30	65.0	9.0	25	116
qsort50	56.4	11.2	35	98
palin25	44.3	10.5	21	67
times10	28.9	7.7	13	48
divide10	24.2	7.8	11	42
log10	41.2	7.8	15	56
ops8	35.4	11.2	21	65
query	56.7	31.9	89	20
Mean	44.5	12.1	28.7	64

Table 2: Performance Comparision

Table 2 shows the comparison of our results with some other WAM based Prolog implementations, i.e. the implementation by D.H.D. Warren [War77] on a DEC-2060 computer, the use of the NCR/32-000 by Fagin [Fag85] where the W-Code is further compiled into 16 bit microinstructions, puters are easily outperformed by SAMP whereas the fastest implementation runs on the VAX using a microcoded W-Code interpreter and beeing slower only in the *query* benchmark.

It is interesting to compare SAMP with another VLIW processor. The QA-2 as described by S. Tomita [Tom86] employs a 256-bit horizontal microinstruction. There are two WAM based Prolog implementations on the QA-2. In the first implementation the W-Code is compiled into QA-2's microcode. The *nrev30* program achieves 45 KLIPS. The other implementation is a microcoded W-Code interpreter where the performance is less than 75% of the former. Thus SAMP is superior in both cases.

6 Conclusion

A high speed Prolog implementation is briefly described. The Warren Abstract Machine is implemented by a microcoded interpreter running on the VLIW processor SAMP. The resulting microprogram has 555 microinstructions. It interprets 74 different instructions of an augmented WAM. The simulator results shows the high performance of 44.5 KLIPS for the Warren Benchmark Set. The new retargetable compiler of the MSS generates the interpreter. The compiler considers alternative execution paths in the hardware structure when allocating different versions for a statement. The high-level MIMOLA description and the retargetability of the compiler simplified the implementation of a virtual machine.

References

[Fag85] B.S. Fagin, Y.N. Patt, V. Srini, A.M. Despain
Compiling Prolog Into Microcode: A Case Study Using the NCR/32-000.
Proc. 18th Ann. Workshop on Microprogramming, Pacific Grove, Dec. 1985, pp.79–88.

8000 Microcode.
ACM SIGMICRO Newsletter, Vol.17, No.4, Dec. 1986, pp.68–74.

[Now86] L. Nowak
SAMP: Entwurf und Realisierung eines neuartigen Rechnerkonzeptes.
PhD thesis and Report 8615 of the Institut für Informatik, University of Kiel, Germany, 1986.

[Now87] L. Nowak
SAMP: A General Purpose Processor Based on a Self-Timed VLIW Structure.
ACM, Comp.Arch.News, Vol.15, No.4, Sep. 1987, pp.32–39.

[Now87a] L. Nowak
Graph Based Retargetable Microcode Compilation in the MIMOLA Design System. Proc. 20th Ann. Workshop on Microprogramming, Colorado Springs, Dec. 1987, pp.126–132.

[Sch88] W. Schenk
Eine Prologimplementierung für einen Rechner sehr großer Befehlsbreite.
Diploma thesis, Institut für Informatik, University of Kiel, Germany, 1988.

[Tic85] E. Tick
Prolog Memory–Referencing Behavior.
Technical Report No. 85-281, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Sep. 1985.

[Tom86] S. Tomita, et. al.
A Computer with Low–Level Parallelism QA-2.
Comp. Arch. News, Vol.14, No.2, 1986, pp.280-289.

[War77] D.H.D. Warren
Applied Logic — Its Use and Implementation as a Programming Tool.
Stanford Research Institute International, Technical Note 290, PhD Thesis, University of Edinburgh, 1977.

[War83] D.H.D. Warren
An Abstract Prolog Instruction Set.
Stanford Research Institute International, Technical Note 309, Oct. 1983.