The Vienna Abstract Machine

Andreas Krall   Ulrich Neumerkel

Institut für Praktische Informatik
Abteilung für Programmiersprachen und Übersetzerbau
Technische Universität Wien
andi@vip.UUCP ulrich@vip.UUCP

Abstract

The Vienna Abstract Machine (VAM) is a Prolog machine developed at the TU Wien. In contrast to the standard implementation technique (Warren Abstract Machine – WAM), an inference in VAM is performed by unifying the goal and head immediately, instead of bypassing arguments through a register interface. We present two implementations for VAM: VAM_{2P} and VAM_{1P}. VAM_{2P} is well suited for an intermediate code emulator (e.g. direct threaded code) which uses two instruction pointers for both goal code and head code. During an inference VAM_{2P} fetches one instruction from the goal code, and one instruction from the head code and executes the combined instruction. More optimization is therefore possible, since information about the calling goal and the head of the clause is available at the same time. VAM performs cheap shallow backtracking, needs less dereferencing and trailing and implements a faster cut. In a Prolog with the occur-check, VAM omits many unnecessary operations. VAM_{1P} is designed for native code compilation. It combines instructions at compile time and supports several optimizations, such as fast last-call optimization. In this paper we present the VAM in detail and compare it with existing machines.

1 Introduction

Five years ago, we began on research in the area of implementation of logic programming languages. A small, slow and portable interpreter [Ge84] and a fast compiler based on the WAM (Warren Abstract Machine [Wa83]) for a commercial Prolog System [Pi84] were developed. With this experience the VIP research project [Op85] was started. Our project developed new interpreter and compiler implementation techniques [Kr87], extensions for meta programming and constraints [Ne88, Ne90], multi user implementations of Prolog with a shared database, database systems [Kü88] etc. One of the results of the project was the design and implementation of the VAM (Vienna Abstract Machine).

In order to outline the major differences between VAM and WAM, we will first present an abstract machine for restricted clauses in Chapter 2. This simplistic model is generalized in Ch. 3 to the basic VAM, focusing on the emulator implementation VAM_{2P} and its basic optimization schemes. Further aspects of VAM are dealt with in Ch. 4. In particular: meta-call, garbage collection, occur-check, delay mechanisms and constraints. Finally, more sophisticated optimizations are presented: the native code model VAM_{1P} and improved calling sequences for a hybrid between VAM_{2P} and VAM_{1P}. Ch. 5 gives a brief comparison with WAM and describes future work on VAM currently under investigation.

```
clause(Head,Goals) --> head(Head), body(Goals).

body(true) --> [c-nogoal].
body(Goals) --> goallist(Goals), [c-lastcall], {Goals \= true}.

goallist(Goal) --> goal(Goal), {Goal \= (_,_)}.
goallist((GoalA,GoalB)) --> goallist(GoalA), [c-call], goallist(GoalB).

head(Head) -->        [F/A],  {functuniv(Head,F,A,L)}, argumentlist(h,L).
goal(Goal) --> [c-goal,F/A],  {functuniv(Goal,F,A,L)}, argumentlist(g,L).

argument(X,Const) --> [X-const,Const], {const(Const)}.
argument(X,Str)   --> [X-struct,F/A], {functuniv(Str,F,A,L)}, argumentlist(X,L).

argumentlist(X,[]) --> [].
argumentlist(X,[E|Es]) --> argument(X,E), argumentlist(X,Es).

functuniv(Funct,F,A,L) :- functor(Funct,F,A), Funct =.. [F|L].
```

Figure 1: Clause representation in VAM for ground clauses

2 A simplified model for ground clauses

Initially, we restrict clauses to those containing no variables at all. While this is not a realistic subset of Prolog for practical applications, it serves to clarify the fundamental differences between WAM and VAM. Later on, we will relax this restriction to programs, where variables are allowed in clauses, provided they will unify with other variables or constants only.

2.1 Representation of clauses

The representation of clauses in VAM intermediate code is very close to their syntactic representation. By and large, terms are translated to a flat prefix code. In a clause three different kinds of codes are used:

control codes, (`c-Any`) are used to embrace goals. A goal starts with `c-goal` $\langle p \rangle$ and either ends with `c-call` if another goal is thereafter or ends with `c-lastcall` if it is the last goal in a clause. If the clause is a fact we have no goal at all denoted by `c-nogoal`.

head codes, (`h-Any`) are used to encode terms in the arguments of a clause's head. The arguments are translated into flat prefix code.

goal codes, (`g-Any`) encode terms in goals. The structure is the same as for head codes.

In Fig. 1 the complete (bijective) mapping between ground clauses and VAM code is defined by a Definite Clause Grammar (DCG).

2.2 Specification of VAM

VAM instructions differ fundamentally from WAM instructions. They can be understood only by their combination at runtime. The *real* instruction set of VAM is the set of all valid combinations of instructions. Taking the translation of Prolog clauses to VAM code and a simple meta-interpreter as input, the VAM could probably be derived automatically by partial evaluation (deduction)—being in the style of [Ku87]. However, the abstract interpreter as well as the complete VAM was designed by hand.

In Fig. 2 an abstract interpreter for VAM code is given. The specification describes the process of unification and (determinate) control in detail, but—similar to [BB83]—it does not explicitly cover backtracking aspects. A program to be interpreted is represented by the `vam_clause/1` facts. A fact `vam_clause([PredName|Cs])` consists of the predicate name and the VAM code translated in Fig. 1. If a predicate consists of several clauses the goal `...,vam_clause([NextPred|NHs]),...` will yield several solutions. Backtracking is therefore implicit in the specification. Later on in Ch. 2.3, in the discussion of actual implementations, backtracking will also be dealt with explicitly.

The procedural behavior of VAM is described by the proof tree of the logic program. First a query is translated like the body of a clause, then the corresponding predicate is fetched and the interpreter is finally called.

The interpreter consists of a (tail recursive) predicate `vam_prove/3` which holds the interpreter state consisting of: the list of remaining head codes, the list of remaining goal codes and a continuation stack for nested calls. The process of proving a goal consists of two major steps corresponding to the different kinds of codes (Ch. 2.1): unification and resolution. By consequence an iteration in the interpreter `vam_prove/3` (a recursive call) can be performed in two ways, either via `unification/3` or `resolution/5` respectively[1]. The state transitions are specified by facts in order to emphasize which states are changed. Before trying to prove these facts, the interpreter takes the first elements of both lists (head and goal code) and combines them (symbolized by the functor `+/2`) in order to pass them to the facts. The effective instructions `HeadCode+GoalCode` are derived by generating all valid combinations of head and goal codes.

`unification(Instruction,DifflistHead,DifflistGoal)` An attempt is made to unify corresponding arguments of head and goal; the remaining codes are passed back to `vam_prove/3`. Combinations such as `h-struct+g-const` are not stated, they simply fail. Note that `unification/3` changes only the two code lists. Later on we will see that unification can be performed by incrementing and adding two pointers. Another observation evident from the specification is that arbitrarily nested structures which occur both in the head and in the goal neither need a *push down stack* nor a counter to unify their arguments since the functors `F/A` do not insert their arity into `vam_prove/3`'s state.

`resolution(Instruction,HeadCode,DifflistGoal,DifflistStack,NextPred)` A clause of `NextPred` is selected by the interpreter (see goal `vam_clause/1`). If a fact in the head was proved and if the body contains another subgoal (`c-nogoal+c-call`), the new goal is selected. The stack is not affected at all. If the head unifies and the goal was the last in the caller's clause (`c-goal+c-lastcall`), the head code will become the new goal code. Again, the stack is not altered (last-call optimization). If the head unifies and there is another goal in the caller's clause (`c-goal+c-call`), then the continuation is pushed onto the stack, the head code becomes the new goal code and the interpreter switches to the new clause's head. The stack needs to be popped if a fact unifies, and if the goal is the last in the caller's clause (`c-nogoal+c-lastcall`). Execution proceeds with the popped continuation. If the stack is empty, the interpreter halts successfully.

There are two straightforward approaches in implementing our specification model:

- Take the `+`'s literally, by combining head and goal at runtime. This abstract machine needs two instruction pointers, henceforth called VAM$_{2P}$. It will lead to the VAM

[1]Note that there is exactly one or no match for a correct goal `vam_prove/3`

emulator using direct threaded code, to be discussed in the next chapter.

- Combine the instructions earlier—at compile time. Only one instruction pointer is therefore required. Such a machine is called VAM$_{1P}$. The instruction pointer can easily be mapped to the program counter of a processor. VAM$_{1P}$ will lead to a subroutine threaded code implementation and further on to an inline code compiler.

When comparing the VAM specification to the ZIP-specification [BB83] we see that ZIP needs an intermediate list for passing arguments. First, their arguments are pushed onto a list and then this list is unified with the head. Although ZIP's instructions are similar to VAM's, they have an implicit operand—the argument stack (refer to Fig. 7). In Fig. 3 some of the cases from unification of arbitrary clauses is given.

Note that PLM [Wa77] behaves in the same way as VAM for ground clauses. But if structured terms with variables are encountered (e.g. `...,p(X/Y),...`) PLM creates always the variables for the molecule on the *global* stack.

The VIP project focused on a compact representation of clauses and we therefore were concerned mainly with VAM$_{2P}$. Yet we are aware of having inline code as an option. In the subsequent sections VAM$_{2P}$ will be presented. VAM$_{1P}$ will be covered later in Ch. 4.5.

2.3 Implementation of VAM$_{2P}$ for ground clauses

2.3.1 Memory model

The basic instructions for ground Prolog clauses are implemented in Fig. 10. The subset of the VAM requires 6 registers and a stack. Two registers are instruction pointers to the code. During unification `goalptr` points to the goal code and `headptr` to the head code. (The stack grows downwards.) A stack frame contains the continuation code pointer (the return address in a procedural language), and the continuation frame pointer (dynamic link). The `stackptr` marks the top of the stack. The `goalframeptr` points to the stack frame of the clause which contains the calling subgoal. The `headframeptr` points to the stack frame of the called clause. The `lastchoiceptr` points to the most recent choice point. A choice point contains pointers to: the previous choice point, the parent stack frame, the calling goal and to the called clause (used to retrieve alternatives).

2.3.2 Unification

During unification one instruction is fetched for the goal arguments and one instruction is fetched for the head arguments. To enable fast decoding of the instructions, the sum of a goal instruction and a head instruction must be unique. Unification for ground clauses is reduced to fetching the next instruction or comparing the instruction arguments.

After unification of the head, at the call of the first subgoal in a clause, the continuation pointers `goalframeptr` and `goalptr` are saved in the clause's stack frame (marked by `headframeptr`). The head pointers are copied to the goal pointers; `goalptr` now points to the first subgoal and the head stack frame has become the stack frame of the calling goal. `headptr` must be set to the called clause thereafter. The simple implementation assumes that the absolute address of the procedure's first clause is stored as an argument of `c-goal`. In a realistic implementation, the macro `clause_address` implements clause indexing. The stack frame for the called clause is allocated by decrementing `stackptr` by `frame_size`. For ground clauses a constant size is sufficient. When there are variables and when we differentiate between facts and rules, `frame_size` depends on the clause and is coded at the start of the clause. The absolute address of an alternative clause is coded at the beginning of a clause too.

```
% unification/3: The unification instructions
%
% unification(Head+Goal, HeadsIn-HeadsOut, GoalsIn-GoalsOut)

unification((h-const)+(g-const), [Const|Hs]-Hs, [Const|Gs]-Gs).
unification((h-struct)+(g-struct), [F/A|Hs]-Hs, [F/A|Gs]-Gs).

% resolution/5: Goal selection
%
% resolution(Head+Goal, Heads, GoalsIn-GoalsOut, StackIn-StackOut, NextPred)

resolution((c-nogoal)+(c-call), [], [c-goal,F/A|Gs]-Gs, St-St, F/A).
resolution((c-goal)+(c-lastcall), [F/A|Hs], []-Hs, St-St, F/A).
resolution((c-goal)+(c-call), [F/A|Hs], Gs-Hs, St-[Gs|St], F/A).
resolution((c-nogoal)+(c-lastcall), [], []-Gs, [[c-goal,F/A|Gs]|St]-St, F/A).

% vam_prove/3: Abstract interpreter
%
% vam_prove(HeadList,GoalList,Stack)

vam_prove([c-nogoal],[c-lastcall],[]).
vam_prove([H|Hs],[G|Gs],St) :-
    unification(H+G,Hs-NHs,Gs-NGs),
    vam_prove(NHs,NGs,St).
vam_prove([H|Hs],[G|Gs],St) :-
    resolution(H+G,Hs,Gs-NGs,St-NSt,NextPred),
    vam_clause([NextPred|NHs]),
    vam_prove(NHs,NGs,NSt).

query(Query) :-
    parse(body(Query),[c-goal,F/A|GoalCode]),   % translation
    vam_clause([F/A|HeadCode]),
    vam_prove(HeadCode,GoalCode,[]).
```

Figure 2: An abstract interpreter for VAM

Again in a realistic implementation, the macro `alternative` is responsible for clause indexing. If alternative clauses exist, a choice point is created. The execution continues with the unification of arguments of the calling goal and the head of the called clause.

2.3.3 Backtracking and cut

On failure, the choice point is popped, the top of stack is adjusted, the alternative clause is selected, a stack frame is allocated and—if another alternative exists—the choice point is pushed back. If the stack frames have the same size (which is usually the case) only the pointer to the alternative clauses must be pushed again. However, because cut occurs more frequently than pushing the choice point a second time, we push the stack frame before the choice point, allowing a faster cut.

```
% extension to DCG for <<interpretation>>

argument(X,Var) --> [X-fstvar,Var].      % assign/initialize
argument(X,Var) --> [X-nxtvar,Var].      % skip/assign/unify
argument(X,_)   --> [X-void].            % skip

% remaining unification instructions
% excerpt from 25 combinations

unification((h-void)+(g-void),Hs-Hs,Gs-Gs).                       % skip
unification((h-fstvar)+(g-void),[HVarNr|Hs]-Hs,Gs-Gs).            % init h-fstvar
unification((h-fstvar)+(g-fstvar),[Var|Hs]-Hs,[Var|Gs]-Gs).      % init both
unification((h-fstvar)+(g-nxtvar),[Var|Hs]-Hs,[Var|Gs]-Gs).      % pass argument
unification((h-const)+(g-fstvar),[Const|Hs]-Hs,[Const|Gs]-Gs). % no trail check
unification((h-const)+(g-nxtvar),[Const|Hs]-Hs,[Const|Gs]-Gs). % trail check
unification((h-nxtvar)+(g-nxtvar),[Var|Hs]-Hs,[Var|Gs]-Gs).      % full unification

unification((h-void)+(g-struct),Hs-Hs,[F/A|Gs]-NGs) :-              % skip goal,
    parse_dl(argument(g,_),[g-struct,F/A|Gs]-NGs).                % init some g-fstvar

unification((h-struct)+(g-fstvar),[F/A|Hs]-NHs,[GVarNr|Gs]-Gs) :-  % no trail check
    parse_dl(argument(h,GVarNr),[h-struct,F/A|Hs]-NHs).
```

Figure 3: Full VAM$_{2P}$specification

2.3.4 Last-call optimization

In ground clauses, last-call optimization is very simple. The goal stack frame is used for the called clause. If the called clause is nondeterminate, the continuation of the callers clause is copied to the stack frame of the called clause. This avoids useless dereferencing when returning from a fact. If the end of a fact is reached (`c-nogoal`), `stackptr` is adjusted and the next instruction code following the goal is fetched. Depending on the fetched code one of three instructions is executed: a `c-cut` instruction (dereferencing `lastchoiceptr` until it becomes greater then the active stack frame), a `c-nogoal` instruction (going up one stack frame) or a `c-goal` instruction.

3 Basic VAM

3.1 Representation of clauses

The control instructions are basically the same as for ground clauses (see Fig. 6). Variables are classified in VAM by their occurrences in heads or goals. For non-void variables (occurring more than once in the clause) the first occurrence is distinguished from the subsequent occurrences. For details refer to Fig. 6. Variables occurring in the head and in subsequent determinate BIPs (Built-In Predicates) are temporary. They are not saved beyond an inference. All other variables are stored in an environment. For example, the predicate `member/2` is translated as follows:

```
member(       T,         [         T   |      _ ]      ).
[member/2,h-fsttmp, 0, h-list, h-nxttmp, 0, h-void, c-nogoal]
member(       X,         [         _  |       Y     ]) :-
[member/2,h-fstvar, 1, h-list, h-void, h-fstvar, 2,
     member(                   X,          Y            ).
      c-goal, member/2, g-nxtvar, 1, g-nxtvar, 2, c-lastcall]
```

`goalptr`	code of caller's goal
`goalframeptr`	stack frame of caller
`headptr`	code of callee's head
`headframeptr`	stack frame of callee
`stackptr`	top of environment stack
`copyptr`	top of copystack
`choiceptr`	top of trail
`trailptr`	top of trail
`lastchoiceptr`	last choice point
`lastcopyptr`	(for trail check only)

variables ...	local variables	d&n d&n
`goalptr'`	continuation code ptr.	d&n
`goalframeptr'`	continuation frame ptr.	d&n
`trailptr'`		n
`copyptr'`	see `lastcopyptr`	n
`headptr'`	alternative	n
`goalptr'`	restart code ptr.	n
`goalframeptr'`	restart frame ptr.	n
`lastchoiceptr'`		n

Figure 4: Registers and stack frames of VAM$_{2P}$

3.2 Memory model

Our implementation of VAM is based on structure copying [Me82]. The representation of dynamic terms is similar to other structure copying implementations. A WAM could use VAMs term representation [Kr88] and *vice versa*. The VAM uses three stacks like [Br84] (refer to Fig. 5). The **environment stack** contains the stack frames which hold local variables and control information. It is either a determinate or a nondeterminate stack frame (choice point), see Fig. 4.

To enable last-call optimization, lists and structures are stored in the **copy stack**. Variable bindings are stored on the **trail**. Space is also needed for temporary variables and the Prolog code (heap). To enable fast comparison of pointer directions the stacks are ordered the way shown in Fig. 5. The basic register set is given in Fig. 4.

3.3 Implementation of resolution and unification

The intermediate code interpreter written in C uses a `switch` statement. The assembler version uses direct threaded code [Bel73]. In the assembler version, the codes for the goal and head instructions are chosen, so that the sum of two instructions is the address of the unification code. The advantage of the parallel unification is that only one decoding step is necessary for two instructions. On a CISC, one `jump` instruction is saved compared to an implementation of the WAM. The following example shows instruction fetch on a CISC.

```
Rn = *goalptr++;      fetch goal instruction
Rn += *headptr++;     add head instruction
jump (Rn);
```

On most RISC-processors there is no auto-increment addressing, but there is normally a subroutine call that stores the return address in a register. The following scheme enables compilation of VAM$_{2P}$ to a mixture of native code and direct threaded code. The goal compiles into 2 to 4 instructions. The head is still in intermediate code.

```
    headaddr = *headptr;
    <goal operand fetch>
    jalr  goalptr, (headaddr[g_code]);

h_code+g_code:
    <head operand fetch + action>
    headptr += opsize;
    jump (goalptr);
```

3.4 Optimizations

The basic VAM_{2P} implements the following optimizations: variable classification, clause indexing and last-call optimization. For the sake of simplicity, the basic VAM_{2P} uses the first argument for clause indexing. Only `g-nxtvar` is used for clause indexing. (Constants can be optimized by program transformation.)

Naïve last-call optimization is performed as described in [Br84]: VAM unifies head and goal and executes subsequent determinate BIPs (e.g. `!/0`). When another goal is encountered (see `c-goal`, vz, p in Fig. 6) the stack frame of the called clause is copied over the caller's frame, if there are no alternatives.

The process of copying moves cells containing pointers. In general, two passes (updating and copying) are required for last-call optimization due to references into the deallocated frame and due to references into the frame to be moved. Restrictions on the following references reduce overheads down to a single copying pass:

References within the head frame Picking away temporaries in the goal prevents their creation throughout an inference.

References to the goal frame References within a frame cause no harm until the last goal. Remaining variables referring to free cells in the goal frame are stored onto the copy stack when encountered during the last goal's inference, similar to WAM's `put_unsafe`.

There are no references within the head frame, so we can hold all head variables (and head temporaries) in registers. Instead of copying an argument into the head frame and recopying it after the inference, all head variables—now held in registers—are placed in the environment. The last-call optimization no longer imposes overheads. The register set is simply stored in place of the old frame. Note that having registers for head variables does not require unsafe variables.

The instruction fetch overhead in an emulator can be reduced by augmenting the instruction set with new folded instructions. This technique is common practice, when only one instruction pointer is required (e.g. WAM emulator). The VAM_{2P} cannot use this technique as exhaustively as the WAM, because a new instruction introduced in the head needs to be encoded N_{goal} times and *vice versa!* On the other hand it is easy to argue that—because VAM_{2P} instructions are folded at runtime—the instruction set is saturated already. Some extensions to list instructions are considered worth the effort. Folded are: (`h-list`, `h-fstvar`), (`h-list`, `h-nxtvar`), (`h-list`, `h-fsttmp`) and (`h-list`, `h-nxttmp`).

4 Extensions and Optimizations

4.1 Meta-call

The meta-call enables dynamic calls to goals by a term. Terms need to be converted into the goal representation. While the WAM passes the structure's arguments into the argument registers, the VAM needs a different approach because VAM executes both goal and head instructions together. A naïve solution implements a meta-call along with the data base that has to support immediate or logical update view:

```
metacall(Goal) :-
    functor(Goal,F,A), functor(NGoal,F,A),
    assert( dummy(NGoal), (retract(dummy(_)), !, NGoal) ),
    dummy(Goal).
```

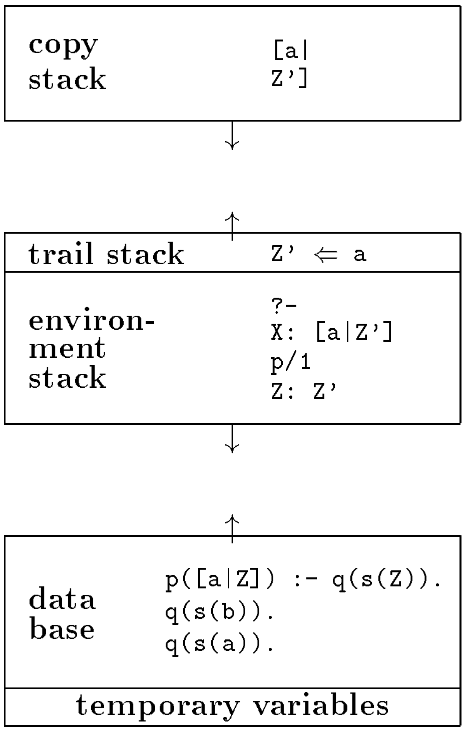

Figure 5: Memory model

A clause "`dummy(goal(X)) :- retract(dummy(_)), !, goal(X).`" is asserted by the goal `metacall(goal(Any))`. Optimizing `metacall/1` in order to avoid the data base operations is straightforward. The term representation is reused as a substitute for an environment. The `goalframeptr` is set to the representation of the term `goal(Any)` on the copy stack, then `goalptr` is set to a dummy goal code consisting of a sequence of `g-nxtvar`s and a closing `c-metacall`. The dummy goal code is reusable for goals with different functors. `c-metacall` is similar to `c-lastcall`. While `c-lastcall` occasionally overwrites the caller's environment, `c-metacall` must not overwrite it because the caller's environment (pointed to by `goalframeptr`) is located on the copy stack representing the term `goal(Any)` at that.

Summarizing, VAM initializes the stack pointers (an operation independent from the number of arguments) and proceeds with decoding the goal and head instructions as usual. Problems concerning treatment of `!/0` within a meta-call are as in WAM.

4.2 Occur-check

Due to efficiency nearly all Prolog implementations perform unification without the occur-check. In general, the occur-check slows down unification by an overhead linear in the terms' sizes, whereas, constant time is required if there is no occur-check. Many unifications deal with terms and variables which were just created by the calling goal. Whilst WAM treats all terms the same way, disregarding the actual pattern of the calling goal, VAM sees the caller as well. Many inferences with structured terms are saved from superfluous occur-checks. In [Bee88] a detailed analysis and examples with no or reduced efforts for occur-checks can be found. VAM behaves with respect to occur-check similar to the extended WAM of [Bee88], designed for hardware implementation. There are cases where VAM performs avoidable occur-checks and trail checks and *vice versa*. Beer's overheads are due to the argument register bottleneck. Our's are due to the inability of VAM$_{2P}$ to let head variables uninitialized beyond the head (similar to WAM). Some of Beer's techniques could be adopted to VAM. A detailed comparison is beyond the scope of this paper.

4.3 Garbage collection

VAM allocates fewer data structures on the copy stack. In VAM, garbage is caused by intermediate structured terms *only.* (If there are no `put_unsafe` instructions). Marking starts from the environment stack. No registers have to be restored from choice points. However, variables within environments may not be completely initialized at marking time because `_-fst*` instructions appear anywhere in a clause. A simple analysis of the whole clause is required, already performed at compile time. For the remaining operations refer to [Br84, PiB85, Ap88].

4.4 Extensions to unification and inference, `freeze/2`

Owing to the many Prolog variations in supporting constraints, a new approach to integrating such extensions was developed. In [Ne90] one of the authors presented *metastructures*, a small extension of Prolog which serves the efficient implementation of meta-logical (e.g. `freeze/2`) and constraint-based extensions. All extensions are defined in Prolog, but efficiency is still comparable to a specialized constraint language. Concerning the abstract machine, we need to execute after the head additional goals triggered by unification. The temporary variables, comparatively few in number, have to be saved, similar to `put_unsafe`'s in WAM. No additional stacks were introduced as in [vC86]. For ordinary programs, a system supporting *metastructures* is at the very most 5% slower than a system without.

4.5 Compilation to VAM_{1P}

Having only one instruction pointer, VAM_{1P} is a model suitable for subroutine threaded or inline code compilation. The instructions to be executed by VAM_{1P} are derived by combining the VAM instructions at compile time. In general a combined instruction has two operands, one belonging to the goal and one belonging to the head. Because combinations with constants can be reduced to true or false during compile time, only instructions where at least one operand is a variable are necessary. Furthermore, the VAM_{1P} does not need temporary variables. The first unification with a temporary variable is delayed until the next unification involving the same variable. The unification partners of the temporary variable are then unified directly with each other. The call of a subgoal is compiled to a cascade of if-instructions for each head of the different clauses. The if-instruction is followed by the unify-instructions for the arguments and a call-instruction (goto-instruction) to the body of the clause. If the clause is a fact, a goto-instruction to the continuation goal follows the unify instructions.

During the compilation of the subgoal-call, the following optimizations should be performed: If the leading parts of alternative heads are the same, they need not be re-evaluated on backtracking. On shallow backtracking, the choice point registers do not need to be restored, because it can be determined at compile time whether the trail or copy stacks will be modified. The pointer to the alternative clauses is contained in the instructions.

If `assert/2` or `retract/2` is executed in the VAM_{1P}, all clauses containing a call to the changed procedure must be recompiled. For a meta-call in VAM_{1P}, code has to be generated for an appropriate entry point. It is also necessary to have a source copy or VAM_{2P} copy of a procedure to compile new clauses and for BIP `clause/2`. Therefore it can be useful to mix compiled and interpreted VAM_{2P} and VAM_{1P} code.

Another problem with the VAM_{1P} as inline code is the size of the generated code (see [De89] for a discussion): If there are n calls to a procedure with m different clauses, $n * m$

unification instructions must be generated, although many instructions will be removed in compensation. There are several solutions if $m * n$ becomes too big:

- Using subroutine threaded code. On some processors subroutine threaded code is faster than direct threaded code.
- Mixing VAM_{1P} and VAM_{2P} code.
- Sharing code. In most cases, the call patterns in the calling subgoals are the same. Therefore the same code can be shared by different clauses if the continuation is saved before start of the unification and variables are renumbered in the same way.
- Inserting a dummy clause which establishes a uniform interface. Here, VAM_{1P} comes close to WAM.

4.6 Last-call optimization

Naïve last-call optimization is performed in VAM_{2P} by updating the references and copying the new stack frame over the old stack frame. This is a time consuming task.

If predicates are static, an interprocedural analysis may derive that the awkward handling of lastcall can be simplified down to allocating variables of the new environment directly in the old. If the variables are ordered correctly (which may not be the case in general) the `headframeptr` is set equal to `goalframeptr`. All instructions are decoded as usual, however, no copying has to be done.

For VAM_{1P}, it is always possible to have the variables for unification in the same stack frame (`headframeptr` and `goalframeptr` are equal). The number of collisions is reduced by reordering argument unification and renumbering variables. Remaining collisions are resolved by temporary variables.

The benefit of last-call optimization is that all instructions can be eliminated, when the operands of the unification are the same. The VIP interpreter uses mixed interpreted VAM_{2P} and VAM_{1P} code. All clauses are compiled to VAM_{2P} code. Additionally the last goal in a determinate recursive clause is compiled into VAM_{1P} code using last-call optimization without copying. This gives fast performance while not wasting memory.

The current VIP assembler interpreter executes at 75 KLIPS on Apollo 3500 (25MHZ 68030), see Fig. 8. 70% of execution time is spent during unification and 30% of the time is spent during resolution (calling the subgoal and making lastcall optimization). 22% of the time is used for instruction fetch and decode (fetching the two codes, adding and jump), 15% of the time is used for operand fetch (loading offsets for variables). Folding instructions and operands speeds up the interpreter to 90 KLIPS. Changing the current tag representation and decoding and improved last call optimization should probably speed up the interpreter to 120 KLIPS.

5 Conclusions

Our experience using VAM for the VIP-system has shown that the VAM-model is a realistic alternative to traditional implementation techniques. The new abstract machine utilizes memory efficiently as well as giving fast execution. The native code variants are comparable to hand coded programs.

5.1 Comparison with WAM

VAM is different from other implementation models w.r.t.: the stack and instruction pointers, the content of stack frames and choice points and the implementation of unification.

Control Instructions			
Mnemonic	*Arguments*	*Position*	*Description*
`c-nogoal`	—	after fact	
`c-goal`	vz, p	first subgoal	fetch pred. p
	p	subsequent goal	
`c-xgoal`	vz, p, $varnr$	first subgoal	fetch p, 1st arg. indexed
	p, $varnr$	subsequent goal	
`c-call`	—	end of a goal	
`c-lastcall`	—	end of clause	lastcall optimization
`c-metacall`	—	end of meta clause	
`c-nbip`	vz, n, $Args$	first subgoal	call nondet. BIP n
	n, $Args$	subsequent subgoal	
`c-dbip`		any subgoal	call det. BIP n
`c-cut`	—		cut choice points

vz ... size of caller's variable frame

Argument Instructions					
Mnemonic		*Arguments*	*Occurrence*	*Description*	*Position*
Head	Goal				
`h-nil`	`g-nil`	—	all	`[]`	any
`h-const`	`g-const`	$aind$		atom	
		nr		integer	
`h-list`	`g-list`	h, t		list `[`h`|`t`]`	
`h-struct`	`g-struct`	f/a, $Args$		structure	
`h-void`	`g-void`	—	once	void	any
`h-fstvar`	`g-fstvar`	$varnr$	first	variable	not last goal
`h-nxtvar`	`g-nxtvar`	$varnr$	subsequent	variable	safe
`h-fsttmp`	—	$tmpnr$	first	temporary	head only
`h-nxttmp`	—	$tmpnr$	subsequent	temporary	head only
—	`g-fstuns`	$varnr$	first	unsafe	last goal only
—	`g-nxtuns`	$varnr$	subsequent	unsafe	1st in lastgoal

Figure 6: VAM$_{2P}$ Instructions

Machine	*yr.*	*Operands*		*Decoding*	*Implicit operands*	*control transfer position*	*instruct. removal*
		Head	Goal				
PLM	77	2	1	`h [g]`	none	prefix	n
ZIP	83	1	1	`g, h`	arg-stack	postfix	y
WAM	83	2	2	`g, h`	none	postfix	y
VAM$_{2P}$	86	1	1	`h+g`	none	prefix	n
VAM$_{1P}$	86	0	2	`g`	none	prefix	y

Figure 7: Comparison of instruction formats

VIP-Version	*old*	folded	new tag
KLIPS	75	90	120
Unification	70	66	60
Resolution	30	34	40
Instruction fetch	22	25	35
Operand fetch	15	0	0

Figure 8: VAM_{2P} timings (*folded* and *new tag* estimated)

In contrast to VAM, the WAM splits the process of inference into a parameter passing and a unification part. To perform an inference, the parameters are passed via argument registers (*put & unify*-instructions); the control is transferred to the called clause, and the parameters in the argument registers are unified with the arguments of the head (*get & unify*-instructions). So WAM goes:

`put`, `put`, ..., `call` $\langle p \rangle$, `get`, `get`, ...

VAM makes `put`s and `get`s at once. It goes:

`c-*goal+c-call` $\langle p \rangle$, `g-Any+h-Any`, `g-Any+h-Any`, ...

Whereas WAM creates data superfluously on the copy stack (heap) for unifying ground structures which are both in goal and head, VAM creates no terms at all for ground programs. VAM, although a structure-copying interpreter, has properties similar to those of structure sharing. The different implementations of inferences influence the memory model, memory utilization and runtime performance.

Because WAM's argument registers must be saved in the choice point, choice point creation and backtracking (especially shallow backtracking [Ti88]) is more expensive than in VAM. On backtracking, VAM has to execute `put+get`-instructions of the goal and the next clause. WAM has to execute `get`-instructions only. WAM's overhead of restoring the arguments is approximately equivalent to the "`put`-overhead" (fetching `g-Any`'s) of VAM. In general, the VAM has fewer trailing and dereferencing operations. A WAM with a separate tag for identifying free and uninitialized variables[Bee88] can yield similar behavior as VAM. However, tag decoding is costly on a conventional processor ([Bee88] is concerned with hardware implementation).

In the VAM, temporary variables cannot be shared between the head and the first subgoal. Variables only occurring in the head and the first goal must be stored as permanent (*local*) in VAM. Therefore in clauses with more than one subgoal the stack frame is larger for the call of the first subgoal provided that WAM can share temporaries (typically 2 to 3 Elements). In determinate clauses with one subgoal VAM's increased stack frame is removed by last-call optimization. If such a clause is nondeterminate, VAM's stack frame is similar in size to WAM's bigger choice point.

Stack trimming poses the same problems in both VAM and WAM.

The VAM needs a smaller copy stack size because VAM has no (or fewer) unsafe variables, and because goal structures need not be stored on the copy stack if they are unified with a void variable of the head or with a matching structure.

The VAM needs no read/write mode (which is also true for an optimized WAM emulator), since the state of the interpreter represents this mode implicitly. Due to the combined decoding, the VAM reduces two decoding steps (`jump`-instructions) to one while the WAM additionally decodes the argument register.

5.2 Further research

The combination of VAM_{2P} and VAM_{1P} seems to be the most promising approach to further improve VAM. Another improvement in the VAM_{2P} spirit under investigation is to delay parts of the head unification unless they are needed. In this way, VAM will be even more similar to a structure sharing interpreter in cases where structure sharing is more efficient than structure copying. If `h-fstvar` unifications are delayed in this way, instruction removal as in a WAM emulator will also be possible for VAM_{2P} without the overhead for shallow backtracking (saving argument registers).

Acknowledgement

We express our thanks to eva Kühn for her work on VIP. Professor Brockhaus always encouraged and supported our project. The anonymous referees helped to improve the paper. Special thanks for revision to Nigel Horspool.

References

[Ap88] Appleby, K. et al., 'Garbage Collection for Prolog Based on WAM', *CACM*, 31(6), 719-741, (JUNE 1988).

[Bee88] Beer, J., 'The Occur-Check Problem Revisited', *JLP* 5(3), (1988).

[Bee89] Beer, J., 'Concepts, Design, and Performance Analysis of a Parallel Prolog Machine', *LNCS*, 404, Springer-Verlag, (OCT. 1989).

[Bel73] Bell, J.R., 'Threaded Code', *CACM*, 16(6), (1973).

[BB83] Bowen, D.L. & Byrd, L.M. & Clocksin, W.F., 'A portable Prolog compiler', Proc. Logic Programm. Workshop, Albufeira, Portugal, (1983).

[Br84] Bruynooghe, M., 'Garbage Collection in Prolog Interpreters', *Implementations of Prolog*, Campbell (ed.), Ellis Horwood, 259-267, (1984).

[Ca87] Carlsson, M., 'Freeze, Indexing and Other Implementation Issues in the WAM', *Proc. 4th Int. Conf. Logic Programm., Melbourne*, Lassez, J.-L. (ed.), MIT Press, (1987).

[De89] Demoen, B. & Mariën, A., 'Inline expansion versus threaded code', `1654@kulcs.kulcs.uucp`, *comp.lang.prolog*, USENET news, (28 APRIL 1989).

[Ge84] Gelbmann, M., *Prolog Interpreter*, Diplomarbeit (M.Thesis), Institut für Praktische Informatik, TU Wien,(1984).

[He89] Hermengildo, M., *"High-Performance Prolog Implementation:" – The WAM and Beyond*, Tutorial at ICLP89 Lissabon, (1989)

[Kr86] Krall, A., 'Comparing Implementation Techniques for Prolog', *VIP TR 1802/86/7*, TU Wien, (1986).

[Kr87] Krall, A., 'Implementation of a High-Speed Prolog Interpreter', *ACM SIGPLAN, Conf. Interpr. and Interpretive Techn.*, 7(7), (1987).

[Kr88] Krall, A., *Analyse und Implementierung von Prologsystemen*, Dissertation TU Wien, (1988).

[Kü88] Kühn, e. & Ludwig, Th., 'VIP-MDBS: A Logic Multidatabase System', *IEEE Int. Symp. on Databases in Parallel and Distributed Systems*, (1988).

[Ku87] Kursawe, P., 'How to Invent a Prolog Machine', *New Gen. Comp.*, 5 (1987) 97-114.

[Me82] Mellish, C.S., 'An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter', *Logic Programming*, Academic Press, (1982).

[Ne88] Neumerkel, U., 'Metastrukturen in Prolog', *Abschlußbericht des Jubiläumsfondsprojektes Nr.2791 der Oesterr. Nationalbank*, (1988); also *VIP TR 1802/88/4*, TU Wien, (1988).

[Ne90] Neumerkel, U., 'Extensible Unification by Metastructures', *Proc. Meta90*, Leuven, Belgium, (1990).

[Op85] Oppitz, M., et al., 'VIP – A Prolog Programming Environment', *TR 1802/85/1*, TU-Wien, (1985).

[Pi84] Pichler, Ch., *Prolog Übersetzer*, Diplomarbeit (M.Thesis), Inst. f. Prakt. Informatik, TU-Wien, (1984).

[PiB85] Pittomvills, E., Bruynooghe, M. & Willems, Y.D. 'Towards a Real Time Garbage Collector for Prolog', *IEEE 1985 Symp. on Logic Programm.*, 185-198, (1985).

[Ti88] Tick, E., *Memory Performance of Prolog Architectures*, Kluwer Acad. Publ., (1988).

[vC86] Caneghem, M. van, *L'Anatomie de Prolog*, InterÉditions, Paris, (1986).

[Wa77] Warren, D.H.D., 'Implementing Prolog – compiling predicate logic programs, Vol. 1 & 2', *D.A.I. Res. Rep. No. 39 & No. 40*, (MAY 1977).

[Wa83] Warren, D.H.D., 'An Abstract Prolog Instruction Set', *TR 309*, SRI Int-l, (1983).

```
MACROS

min(a,b)                          a < b ? a : b
deref(ptr)                        ptr = *ptr
write_frame(ptr, a, b)            *ptr = a; *(ptr+1) = b
read_frame(ptr, a, b)             a = *ptr; b = *(ptr+1)
copy_frame(scr, dest)             *dest = *scr; *(dest+1) = *(scr+1)
clause_address(addr)              addr
frame_size(ptr)                   *(ptr-2)
alternative(ptr)                  *(ptr-1)
push_choicepoint(s, l, f, g, h)   *(--s) = h; *(--s) = g; *(--s) = f; *(--s) = l; l = s
pop_choicepoint(l, f, g, h)       h = *(l+3); g = *(l+2); f = *(l+1); l = * nl
```

Figure 9: Makros for VAM_{2P} interpreter

```
for(;;)
switch(*headptr++ + *goalptr++) {
case c_goal+c_call:
  write_frame(headframeptr, goalptr, goalframeptr);
  goalframeptr = headframeptr;
  goalptr = headptr;
  headptr = clause_address(*goalptr++);
  headframeptr = stackptr -= frame_size(headptr);
  if (alternative(headptr))
    push_choicepoint(stackptr, lastchoiceptr, goalframeptr, goalptr, headptr);
  continue;
case c_cut+c_lastcall: case c_cut+c_call:
  if (lastchoiceptr < headframeptr)
    {deref(lastchoiceptr); stackptr = headframeptr;}
  goalptr--;
  continue;
case c_goal+c_lastcall:
  if (lastchoiceptr < goalframeptr) /* no tail recursion */
    {copy_frame(goalframeptr, headframeptr); goalframeptr = headframeptr;}
  goalptr = headptr;
  headptr = clause_address(*goalptr++);
  headframeptr = stackptr -= frame_size(headptr);
  if (alternative(headptr))
    push_choicepoint(stackptr, lastchoiceptr, goalframeptr, goalptr, headptr);
  continue;
case c_nogoal+c_lastcall:
  read_frame(goalframeptr, goalptr, goalframeptr);
case c_nogoal+c_call:
  stackptr = min(lastchoiceptr, goalframeptr);
  for(;;) {
    switch(*goalptr++) {
      case c_cut:
        while (lastchoiceptr < goalframeptr) deref(lastchoiceptr);
        stackptr = goalframeptr;
        continue;
      case c_nogoal:
        read_frame(goalframeptr, goalptr, goalframeptr);
        stackptr = min(lastchoiceptr, goalframeptr);
        continue;
      case c_goal:
        headptr = clause_address(*goalptr++);
        headframeptr = stackptr -= frame_size(headptr);
        if (alternative(headptr))
          push_choicepoint(stackptr, lastchoiceptr, goalframeptr, goalptr, headptr);
    }
    break;
  }
case h_empty+g_empty: case h_list+g_list:   continue;
case h_const+g_const:   case h_structure+g_structure:
    if (*goalptr++ == *headptr++) continue; /* functor */
default: /* fail */
    pop_choicepoint(lastchoiceptr, goalframeptr, goalptr, headptr);
    stackptr = min(lastchoiceptr, goalframeptr);
    headptr = alternative(headptr);
    headframeptr = stackptr -= frame_size(headptr);
    if (alternative(headptr))
        push_choicepoint(stackptr, lastchoiceptr, goalframeptr, goalptr, headptr);
}
```

Figure 10: A simple VAM$_{2P}$ for ground clauses with `!/0`