A Progress Report on Incremental Global Compilation of Prolog

Andreas Krall and Thomas Berger
Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8
A-1040 Wien
{andi,tb}@mips.complang.tuwien.ac.at

Abstract

Traditional native code generating Prolog compilers with global analysis compile programs as a whole and do not support the data base builtin-predicates `assert` and `retract`. In this paper we present a scheme to both enable global analysis and incremental compilation. This incremental compiler is based on the Vienna Abstract Machine (VAM). A version of the VAM, the VAM_{AI}, is used as an abstract machine for abstract interpretation. The VAM_{AI} does the data flow analysis by a factor of two hundred faster than the previous used meta interpreter written in Prolog. This fast execution together with a compact representation of the intermediate code makes incremental global compilation feasible. Preliminary results of intermediate code size, size of generated machine code, compile time and run time are presented.

1 Introduction

The development of the Vienna Abstract Machine (VAM) started in 1985. The VAM has been developed as an alternative to the Warren Abstract Machine (WAM). The aim was to eliminate the parameter passing bottleneck of the WAM. The development started with an interpreter [Kra87] which led to the VAM_{2P} [KN90]. Partial evaluation of the call led to the VAM_{1P} which is well suited for machine code compilation [KB92]. The first compiler was a prototype implemented in Prolog without any global analysis. This compiler was enhanced by global analysis. It was further on modified to support incremental compilation. This prototype implementation was limited by the single assignment nature of Prolog, used `assert` and `retract` to manipulate global information and was therefore very slow. So we designed the VAM_{AI}, an abstract machine for abstract interpretation, and implemented the whole incremental compiler in the programming language C.

Section 2 introduces the Vienna Abstract Machine with its two versions VAM_{2P} and VAM_{1P}. Section 3 shows how the VAM can be modified to an abstract machine for abstract interpretation and code generation. Section 4 presents some preliminary results about the size and efficiency of the incremental compiler.

2 The Vienna Abstract Machine

2.1 Introduction

The VAM has been developed at the TU Wien as an alternative to the WAM. The WAM divides the unification process into two steps. During the first step the arguments of the calling goal are copied into argument registers and during the second step the values in the argument registers are unified with the arguments of the head of the called predicate. The VAM eliminates the register interface by unifying goal and head arguments in one step. The VAM can be seen as a partial evaluation of the call. There are two variants of the VAM, the VAM_{1P} and the VAM_{2P}.

A complete description of the VAM_{2P} can be found in [KN90]. Here we give a short introduction to the VAM_{2P} which helps to understand the VAM_{1P} and the compilation method. The VAM_{2P} (VAM with two instruction pointers) is well suited for an intermediate code interpreter implemented in C or in assembly language using direct threaded code [Bel73]. The goal instruction pointer points to the instructions of the calling goal, the head instruction pointer points to the instructions of the head of the called clause. During an inference the VAM_{2P} fetches one instruction from the goal, one instruction from the head, combines them and executes the combined instruction. Because information about the calling goal and the called head is available at the same time, more optimizations than in the WAM are possible. The VAM features cheap backtracking, needs less dereferencing and trailing and has smaller stack sizes.

The VAM_{1P} (VAM with one instruction pointer) uses one instruction pointer and is well suited for native code compilation. It combines instructions at compile time and supports additional optimizations like instruction elimination, resolving temporary variables during compile time, extended clause indexing, fast last-call optimization, and loop optimization.

2.2 The VAM_{2P}

Like the WAM, the VAM_{2P} uses three stacks (see fig. 1). Stack frames and choice points are allocated on the environment stack, structures and unbound variables are stored on the copy stack, and bindings of variables are marked on the trail. The intermediate code of the clauses is held in the code area. The machine registers are the `goalptr` and `headptr` (pointer to the code of the calling goal and of the called clause respectively), the `goalframeptr` and the `headframeptr` (frame pointer of the clause containing the calling goal and of the called clause respectively), the top of the environment stack (`stackptr`), the top of the copy stack (`copyptr`), the top of the trail (`trailptr`), and the pointer to the last choice point (`choicepntptr`).

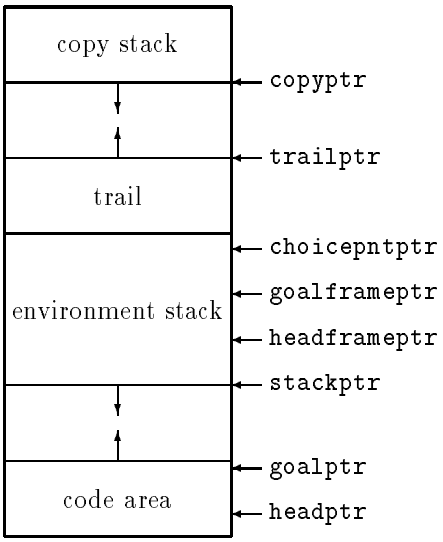

Figure 1: VAM data areas

Values are stored together with a tag in one machine word. We distinguish integers, atoms, nil, lists, structures, unbound variables and references. Unbound variables are allocated on the copy stack to avoid dangling references and the unsafe variables of the WAM. Furthermore it simplifies the check for the trailing of bindings. Structure copying is used for the representation of structures.

Variables are classified into void, temporary and local variables. Void variables occur only once in a clause and need neither storage nor unification instructions. Different to the WAM, temporary variables occur only in the head or in one subgoal, counting a group of builtin predicates as one goal. The builtin predicates following the head are treated as if belonging to the head. Temporary variables need storage only during one inference and can be held in registers. All other variables are local and are allocated on the environment stack. During an inference the variables of the head are held in registers. Prior to the call of the first subgoal the registers are stored in the stack frame. To avoid initialization of variables we distinguish between their first occurrence and further occurrences.

The clauses are translated to the VAM_{2P} abstract machine code (see fig. 2). This translation is simple due to the direct mapping between source code and VAM_{2P} code. During run time a goal and a head instruction are fetched and the two instructions are combined. Unification instructions are combined with unification instructions and resolution instructions are combined with termination instructions. A different encoding is used for goal unification instructions and head unification instructions. To enable fast encoding the instruction combination is solved by adding the instruction codes and, therefore, the sum of two instruction codes must be unique. The C statement

```
switch(*headptr++ + *goalptr++)
```

implements this instruction fetch and decoding.

variables	local variables
`goalptr'`	continuation code pointer
`goalframeptr'`	continuation frame pointer

Figure 3: stack frame

`trailptr'`	copy of top of trail
`copyptr'`	copy of top of copy stack
`headptr'`	alternative clauses
`goalptr'`	restart code pointer (VAM_{2P})
`goalframeptr'`	restart frame pointer
`choicepntptr'`	previous choice point

Figure 4: choice point

2.3 The VAM_{1P}

The VAM_{1P} has been designed for native code compilation. A complete description can be found in [KB92]. The main difference to the VAM_{2P} is that instruction combination is done during compile time instead of run time. The representation of data, the stacks and

unification instructions	
`const C`	integer or atom
`nil`	empty list
`list`	list (followed by its arguments)
`struct F`	structure (followed by its arguments)
`void`	void variable
`fsttmp Xn`	first occurrence of temporary variable
`nxttmp Xn`	subsequent occurrence of temporary variable
`fxtvar Vn`	first occurrence of local variable
`nxtvar Vn`	subsequent occurrence of local variable
resolution instructions	
`goal P`	subgoal (followed by arguments and call/lastcall)
`nogoal`	termination of a fact
`cut`	cut
`builtin I`	builtin predicate (followed by its arguments)
termination instructions	
`call`	termination of a goal
`lastcall`	termination of last goal

Figure 2: VAM_{2P} instruction set

stack frames (see fig. 3) are identical to the VAM_{2P}. The VAM_{1P} has one machine register less than the VAM_{2P}. The two instruction pointers `goalptr` and `headptr` are replaced by one instruction pointer called `codeptr`. Therefore, the choice point (see fig. 4) is also smaller by one element. The pointer to the alternative clauses now directly points to the code of the remaining matching clauses.

Due to instruction combination during compile time it is possible to eliminate instructions, to eliminate all temporary variables and to use an extended clause indexing, a fast last-call optimization and loop optimization. In WAM based compilers abstract interpretation is used to derive information about mode, type and reference chain length. Some of this information is locally available in the VAM_{1P} due to the availability of the information of the calling goal.

All constants and functors are combined and evaluated to true or false. For a true result no code is emitted. All clauses which have an argument evaluated to false are removed from the list of alternatives. In general no code is emitted for a combination with a void variable. In a combination of a void variable with the first occurrence of a local variable the next occurrence of this variable is treated as the first occurrence.

Temporary variables are eliminated completely. The unification partner of the first occurrence of a temporary variable is unified directly with the unification partners of the further occurrences of the temporary variable. If the unification partners are constants, no code is emitted at all. Flattened code is generated for structures. The paths for unifying and copying structures is split and different code is generated for each path. This makes it possible to reference each argument of a structure as offset from the top of the copy stack or as offset from the base pointer of the structure. If a temporary variable is contained in more than one structure, combined unification or copying instructions are generated.

All necessary information for clause indexing is computed during compile time. Some alternatives are eliminated because of failing constant combinations. The remaining alternatives are indexed on the argument that contains the most constants or structures. For compatibility reasons with the VAM_{2P} a balanced binary tree is used for clause selection.

The VAM_{1P} implements two versions of last-call optimization. The first variant (we call it post-optimization) is identical to that of the VAM_{2P}. If the determinism of a clause can be determined during run time, the registers containing the head variables are stored in the callers stack frame. Head variables which reside in the stack frame due to the lack of registers are copied from the head (callee's) stack frame to the goal (caller's) stack frame.

If the determinism of a clause can be detected during compile time, the caller's and the callee's stack frames are equal. Now all unifications between variables with the same offset can be eliminated. If not all head variables are held in registers reading and writing variables must be done in the right order. We call this variant of last-call optimization pre-optimization. This optimization can be seen as a generalization of recursion replacement by iteration to every last-call compared to the optimization of [Mei91].

Loop optimization is done for a determinate recursive call of the last and only subgoal. The restriction to a single subgoal is due to the use of registers for value passing and possible aliasing of variables. Unification between two structures is performed by unifying the arguments directly. The code for the unification of a variable and a structure is split into unification code and copy code.

3 The Incremental Compiler

The compilation of a Prolog program is carried out in five passes. A clause is read in by the built-in predicate `read` and transformed to term representation. The compiler first translates the term representation into VAM_{AI} intermediate code. Incremental abstract interpretation is executed on this intermediate code and the code is annotated with type, mode, alias and dereferencing information. The VAM_{AI} intermediate code is traversed again, compiled to VAM_{1P} code and expanded on the fly to machine code. The last step is instruction scheduling of the machine code and the patching of branch offsets and address constants.

3.1 Abstract Interpretation

Information about types, modes, trailing, reference chain length and aliasing of variables of a program can be inferred using abstract interpretation. Abstract interpretation is a technique of describing and implementing global flow analysis of programs. It was introduced by [CC77] for dataflow analysis of imperative languages. This work was the basis of much of the recent work in the field of logic programming [AH87] [Bru91] [Deb92] [Mel85] [RD92] [Tay89]. Abstract interpretation executes programs over an abstract domain. Recursion is handled by computing fixpoints. To guarantee the termination and completeness of the execution a suitable choice of the abstract domain is necessary. Completeness is achieved by iterating the interpretation until the computed information does not change. Termination is assured by limiting the size of the domain. The previous cited systems all are meta-interpreters written in Prolog and very slow.

A practical implementation of abstract interpretation has been done by Tan and Lin [TL92]. They modified a WAM emulator implemented in C to execute the abstract operations on the abstract domain. They used this abstract emulator to infer mode, type and alias information. They analysed a set of small benchmark programs in few milliseconds which is about 150 times faster than the previous systems.

3.2 The VAM_{AI}

The VAM_{AI} is an abstract machine developed for the quick computation of dataflow information (i.e. types, modes and reference chain length for local variables) of Prolog programs. It has been developed on the basis of the VAM_{2P} and benefits from the fast decoding mechanism of this machine. The inferred dataflow information is stored directly in the intermediate code of the VAM_{AI}. We choose the VAM as the basis for an abstract machine for abstract interpretation because it is much better suited than the WAM. The parameter passing of the WAM via registers and storing registers in a backtrack point slows down the interpretation, since abstract interpretation has no backtracking. Furthermore, in the WAM some instructions are eliminated so that the relation between argument registers and variables is sometimes difficult to determine. The translation to a VAM_{2P} like intermediate code is simpler and faster than WAM code generation. Furthermore, we needed the VAM_{2P} intermediate code for the generation of the VAM_{1P} instructions.

Our goal is to gather information about mode, type and reference chain length. We do not extract trailing information because the VAM does trail more seldom than the WAM. Each of the types of the domain additionally contains information about the reference chain length. We distinguish between 0, 1 and unknown for the reference chain length. Each type represents a set of terms:

any	is the top element of our domain
free	describes an unbound variable and contains a reference to all aliased variables
bound	describes non-variable terms
atomic	is the supertype of nil, atom and integer
list	is a non empty list (it contains the types of its arguments)
struct	is a term (information about the functor and arguments is contained)
nil	represents the empty list
atom	is the set of all atoms
integer	is the set of all integer numbers

Possible infinite nesting of compound terms makes the handling of the types list and struct difficult. To gather useful information about recursive data structures we introduce a recursive type which contains also the information about the termination type.

We use a top-down approach for the analysis of the desired information. We handle different calls to the same clause separately to get the exact types. The gathered information is a description of the living variables. Recursive calls of a clause are computed until a fixpoint for the gathered information is reached. If there exists already information about a call for a caller and the new

gathered information is more special than the previously derived, i.e. the union of the old type and the new type is equal to the old type, we stop the interpretation of this call to the clause. The gathered information has then reached a fixpoint.

With a short example we want do demonstrate the abstract interpretation with VAM_{AI}. Fig. 5 shows a simple Prolog program part, and a simplified view of its code duplication for the representation in the VAM_{AI} intermediate code.

Prolog program:

$A_1 :- B^1$
$B_1 :- C^1$
$B_2 :- B^2, C^2$
$C_1 :- true$

Code representation:

$A_1^1 :- B^1$
$B_1^1 :- C^1$
$B_2^1 :- B^2, C^2$
$B_1^2 :- C^1$
$B_2^2 :- B^2, C^2$
$C_1^1 :- true$
$C_1^2 :- true$

Figure 5: Prolog program part and its representation in VAM_{AI}

The procedure B has two clauses, the alternatives B_1 and B_2. As we can see in the example the codes for the Procedures B and C are duplicated because both procedures are called twice in this program. This code duplication leads to more exact types for the variables, because the dataflow information input might be different (more or less exact) for different calls of the same procedure in a program. We start abstract interpretation at the beginning of the program with the clause A_1^1. The domains of the variables in the subgoal B^1 are determined by the inferable dataflow information from the two clauses B_1^1 and B_2^1. After the information for both clauses is computed the abstract interpretation is finished because there is no further subgoal for the first clause A_1. To be conservative we suppose that both B_1^1 and B_2^1 could be reached during program execution, therefore we have to union the derived dataflow information sets for the alternative clauses of procedure B. For B_1^1 we only have to derive the information from C_1^1 because it is the only subgoal for B_1^1. For B_2^1 there exists a recursive call for B, named B^2 in the example. Recursion in abstract interpretation is handled by computing a fixpoint, i.e. we interpret the recursive call as long as the derived data information changes. After the fixpoint is reached, we can stop the computation for the recursive call. The dataflow information for the recursion is assigned to the clauses B_1^2 and B_2^2. After all inferable information is computed for a clause, it is stored directly into the intermediate code. So we can use the same intermediate code efficiently in the code generation pass of the compiler.

Fig. 6 shows the data types for the intermediate representation of the procedures. The intermediate code is stored as a contiguous block of short and long integer numbers.

```
typedef struct CLAUSE {
        SHORT          var_count;
        SHORT          h_temp_count;
        struct CLAUSE *next_clause;
        CODEPTR        clause_code;
                       }CLAUSE;

typedef struct CLAUSEPAR {
        CLAUSE            *clause;
        struct CLAUSEPAR *next_clause_par;
        struct APROC      *parent_proc;
                          }CLAUSEPAR;

typedef struct APROC {
        CLAUSEPAR    *clause_ptr;
        struct APROC *next_proc;
        MWPTR         functor;
                     } APROC;
```

Figure 6: data types for the intermediate code for procedures

The representation for the arguments of a Prolog term is the same for VAM_{AI} (see fig. 7) and VAM_{2P} with only the following exceptions:

- The intermediate code for the first occurrence of variables (`fstvar` and `fsttmp`) has been removed, because the order of variables can be changed due to optimizations.
- Local variables have two additional information fields in their intermediate code, the actual domain of the variable and the reference chain length.
- The argument of a temporary variable contains an offset which references this variable in a global table. The global table contains pointers to the intermediate code of the unification partner of the temporary variable.
- The intermediate code `lastcall` has been removed because last-call optimization makes no sense in abstract interpretation. Instead the intermediate code `nogoal` indicates the end of a clause. When

this instruction is executed the computation continues with the next alternative clause.

- The intermediate code `goal` got an additional argument, a pointer to the end of this goal, that is the instruction following the call.
- The instruction `const` has been split into `integer` and `atom`.

Another significant difference between the two abstract machines concerns the data area, i.e. the stacks. While the VAM_{2P} needs three stacks, in VAM_{AI} a modified environment stack is sufficient. Fig. 8 shows a stack frame for the environment stack from the VAM_{AI}. Note that there exists no choice point during the abstract interpretation because all alternatives for a call are considered for the result of the computation. Therefore there is no backtracking in the abstract interpretation (that means also that variables need not be trailed and we do not need the trail in the VAM_{AI}).

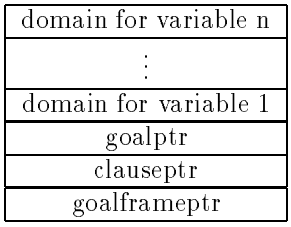

Figure 8: structure of the stack frame

The stack frame contains the actual informations for all the local variables of a clause. The `goalptr` points to the intermediate code of a goal (it is used to find the continuation after a goal has been computed), the `clauseptr` points to the head of the next alternative clause for the called procedure, and `goalframeptr` points to the stack frame of the calling procedure.

Fig. 9 is a detailed description of the stack entry for a local variable.

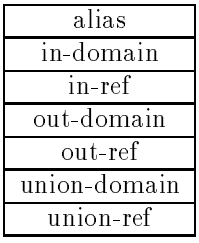

Figure 9: a local variable on the stack

If two unbound variables are unified, one of them gets a reference to the other one. We call those variables aliased. In our abstract interpretation we handle aliasing with a separate field on the stack (`alias`). It possibly contains a pointer to the entry of an other (aliased) variable entry on the stack. Through this field all aliased variables are connected to a chain ending with `nil`. The other fields of an aliased variable are unimportant, they inherit their domains from the last variable in the chain. During computation of the dataflow information for a variable three different information sets are stored on the stack:

- The in-set contains all information which has been gathered for the variable before the computation has entered the actual goal
- The out-set contains all actual information at any state of computation, it is the working set, its content changes at any occurrence of the variable in the intermediate code.
- The union-set gathers the inferred information for each alternative of a call; after all alternatives are computed it contains the type union for all alternatives.

When the abstract interpretation computes a new clause a stack frame has to be allocated on the stack. All local variables are initialized with the type `empty`. During the abstract unification of a goal (the caller) and a head (the callee) only the out-sets are used as working sets. They are changed whenever the inferred information becomes more exact. For a goal information inferred by alternative clauses is gathered in the union-sets by computing the domain union (e.g. the union for `integer`, `atom` and `nil` is `atomic`), so the union of the alternative matching clauses produces more general sets. The in-set is used to keep the information between two goals. After each alternative the out-sets (the working sets) are initialized with the content of the in-sets. During the abstract unification the content of the in-set is compared with the domain field of the intermediate code of a local variable and overwrites this field if the information has become more general since the last computation of this variable. In this way we can also detect if a fixpoint has been reached for a recursive procedure call. We can stop abstract interpretation of a clause if for all variables of a goal the information contained in their domain fields is more general or equal to the in-set of the variables.

Incremental abstract interpretation start the local analysis with all the callers of the modified procedure and interpret the intermediate code of all dependent procedures. Interpretation is stopped, if the derived domains are equal to the original domains (those derived by the previous analysis). If the domain has not changed new code is only generated for the changed part of the program. If the domain has been changed and the new

unification instructions	
`int I`	integer
`atom A`	atom
`nil`	empty list
`list`	list (followed by its two arguments)
`struct F`	structure (functor)(followed by its arguments)
`void`	void variable
`temp Xn`	temporary variable (offset)
`local Vn,Dn,Rn`	local variable (offset, domain, reference chain length)
resolution instructions	
`goal P,O,Tc`	subgoal (procedure pointer, end of goal, number of temporaries)
`nogoal`	termination of a clause
`cut`	cut
`builtin I(,Tc)`	built-in predicate (built-in number, opt. number of temporaries)
termination instructions	
`call`	termination of a goal

Figure 7: VAM_{AI} instruction set

domain is a subtype of the old domain, the previously generated code fits for the changed program part. The information derived for the program before the change is less exact than the possibly derivable information. Incremental interpretation can stop now. If the new domain is instead a supertype of the old one the assertion of the new clause made the possible information less precise and the old code part at this program point wrong for the new program. Therefore incremental abstract interpretation must further analyse the callers of the clause.

The retraction of a clause has a similar effect like the assertion of a new one. The only difference is, that there might not be an information loss which makes the previously generated code of the callers of the calling clauses wrong. Therefore it is sufficient to start the incremental analysis with the calling clauses of the changed procedure and interpret top-down until the derived information is equal to the previously inferred one.

3.3 Compilation to Machine Code

The abstract interpretation pass has annotated a part of the VAM_{AI} intermediate representation with type, mode and dereferencing information. It must be translated to machine code. The first occurrences of temporary variables are replaced by their unification partner. If temporary variables occur in two or more structures which have to be unified with local variables, the arguments are reordered, so that these temporary variables can be identified as an offset from the top of the copy stack. The intermediate representation is traversed and translated to machine code instructions representing VAM_{1P} instructions.

The next step is instruction scheduling. Our scheduler is based on list scheduling [War90]. This is a heuristic method which yields nearly optimal results. First the basic block and the dependence graph for each basic block is determined. The alias information is used to recognize the independence of load and store instructions. Then the instructions are reordered to fill the latency of load instructions and to fill the delay slot of a branch instruction starting with the instructions on the longest path to the end of the basic block. After scheduling the branch offsets and address constants are updated. For this purpose the relocation information stored in the VAM_{AI} intermediate representation is used. So the address changes resulting from scheduling and from incremental compilation can be handled together.

4 Results

Before developing the VAM_{AI} we developed a prototype compiler in Prolog. Due to the single assignment nature of Prolog the information about the program has to be stored in the data base using `assert`. The prototype compiler is implemented as a deterministic recursive procedure so that it was possible to store intermediate representations in Prolog data structures. The problem is that this structures are stored on the copy stack, destructive assignment must be replaced by copying part of the structures and that our Prolog interpreter does not support garbage collection. So this interpreter was very slow and needed a stack size greater than 32 MB when compiling a program which was bigger than 50 clauses. So it is evident that the compiler based on the

test	VAM_{2P} bytes	VAM_{2P} scaled	VAM_{AI} scaled	VAM_{1P}opt scaled	VAM_{1P} scaled
det. append	288	1	3.63	9.96	11.6
naive reverse	380	1	3.59	11.3	13.5
quicksort	764	1	2.65	9.95	11.2
8-queens	536	1	2.95	8.25	9.00
serialize	1044	1	3.33	15.7	19.1
differentiate	1064	1	8.37	28.4	62.7
query	2084	1	0.89	3.13	6.50
bucket	996	1	1.96	9.75	16.9
permutation	296	1	2.77	6.21	7.61

Table 1: code size of intermediate representations

VAM_{AI} is on average more than a factor of two hundred faster than the prototype compiler (see table 2).

test	Prolog ms	Prolog scaled	VAM_{AI} scaled
det. append	648	1	1115
naive reverse	789	1	639
quicksort	977	1	264
8-queens	815	1	241
serialize	1630	1	206
differentiate	2122	1	25
query	781	1	194
bucket	923	1	230
permutation	732	1	503

Table 2: global analysis time, factor of improvement

It is difficult to compare the sizes needed for the data structures. The Prolog interpreter has a built-in predicate which takes a list of assembly language instructions, translates it to machine code, resolves the branches and stores the machine code in the code area. The size of the machine code generated without global analysis is up to a factor of two bigger than the optimized code. So the size of the generated machine code is in both cases identical. The data structures stored using `assert` are about two times as big as the data stored by the VAM_{AI}. The stack use of the VAM_{AI} is neglectable compared to that of the prototype compiler.

Comparing the VAM_{2P} interpreter with the VAM_{AI} it shows that the size of the generated machine code is about ten times bigger than the internal representation of the VAM_{2P} (see table 1). The annotated VAM_{AI} intermediate code is about three times bigger than the simple VAM_{2P} intermediate code. For a comparison of the execution time (see table 3). On average the compiled code is about seven times faster than the interpreted code. The VAM_{2P} interpreter is about the same speed as the SICStus byte code emulator.

test	VAM_{2P} ms	VAM_{1P} scaled	VAM_{1P}opt scaled
det. append	0.25	26.1	26.1
naive reverse	4.17	19.3	20.0
quicksort	6.00	7.23	10.3
8-queens	65.4	12.38	13.5
serialize	3.90	5.76	6.84
differentiate	1.14	6.32	8.14
query	41.7	7.58	9.70
bucket	247	5.02	5.24
permutation	4023	5.08	5.34

Table 3: factor of execution time improvement

5 Conclusion

We presented the VAM_{AI}, an abstract machine for abstract interpretation. This abstract machine has a very compact representation and reduces the analysis time of programs by a factor of two hundred compared to meta interpreters written in Prolog. This fast analysis and the storage of additional information enables the incremental global compilation of Prolog.

Acknowledgement

We express our thanks to Anton Ertl, Franz Puntigam and the anonymous referees for their comments on earlier drafts of this paper.

References

[AH87] Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[Bel73] James R. Bell. Threaded code. *CACM*, 16(6), June 1973.

[Bru91] Maurice Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic programming*, 10(1), 1991.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth Symp. Priciples of Programming Languages*. ACM, 1977.

[Deb92] Saumya Debray. A simple code improvement scheme for Prolog. *Journal of Logic Programming*, 13(1), 1992.

[KB92] Andreas Krall and Thomas Berger. Fast Prolog with a VAM_{1P} based Prolog compiler. In *PLILP'92*, LNCS. Springer 631, 1992.

[KN90] Andreas Krall and Ulrich Neumerkel. The Vienna abstract machine. In *PLILP'90*, LNCS. Springer, 1990.

[Kra87] Andreas Krall. Implementation of a high-speed Prolog interpreter. In *Conf. on Interpreters and Interpretative Techniques*, volume 22(7) of *SIGPLAN*. ACM, 1987.

[Mei91] Micha Meier. Compilation of compound terms in Prolog. In *Eighth International Conference on Logic Programming*, 1991.

[Mel85] Christopher S. Mellish. Some global optimizations for a Prolog compiler. *Journal of Logic Programming*, 2(1), 1985.

[RD92] Peter Van Roy and Alvin M. Despain. High-performance logic programming with the Aquarius Prolog compiler. *IEEE Computer*, 25(1), 1992.

[Tay89] Andrew Taylor. Removal of dereferencing and trailing in Prolog compilation. In *Sixth International Conference on Logic Programming*, Lisbon, 1989.

[TL92] Jichang Tan and I-Peng Lin. Compiling dataflow analysis of logic programs. In *Conference on Programming Language Design and Implementation*, volume 27(7) of *SIGPLAN*. ACM, 1992.

[War90] Henry S. Warren. Instruction scheduling for the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1), 1990.