Register Allocation in WAM

Luděk Matyska, Adriana Jergová, David Toman
Institute of Computer Science
Masaryk University
601 77 Brno, Czechoslovakia
lumat@CSPUNI12.BITNET

Abstract

Paper presents a new register allocation strategy for WAM based compilation of Prolog. The proposed strategy is based on the computation of optimal order of the unification of individual head arguments. The unification of head arguments is interspersed with the execution of *inline calls* and with the preparation of arguments for the *out-of-line call.* Moreover, it is shown that the problem of register allocation in WAM is at least NP-Complete in general.

1 Introduction

Almost all current successful Prolog implementations on traditional computer architectures are based on the Warren Abstract Machine (WAM) architecture and its instruction set [14] (for a good introduction see, e.g., [1]). Many improvements to the WAM have been suggested and also the problem of efficient code generation was extensively studied from several perspectives:

- program transformation (e.g., unfold/fold transformations [5], global flow analysis [15])
- mode and type inference [4, 6, 16]
- clause indexing [2], and
- register allocation [3, 8]

Although current interest in compiler implementation is focused mostly on the analysis of global properties of Prolog programs, the generation of efficient code for individual clauses is still important, esp. in connection with dynamic (i.e., assertable/retractable) procedures [10]. Paper presents a new register allocation strategy for compilation of individual Prolog clauses. The main problem addressed is "which register should be used for which variable to minimize the number of instructions generated".

Several authors have suggested methods for optimal register allocation [3, 8, 9, 13, 17]. The strict left to right order for the treatment of individual head arguments, implicitly present in the original WAM [14], needs not to be relied upon, and more efficient schemes with reordered unification were also proposed [8, 9, 17]. Nevertheless, none of these methods is optimal in all cases. In the first part of the present paper the problem of register allocation in WAM is shown to be at least NP-Complete. In the second part special subcases of the general problem are discussed, and for them the optimal register allocation strategy is presented.

2 Terminology and Basic Notions

Notation of Janssens et al. [8] will be followed throughout this paper, i.e., variables begin in upper case, permanent variables are denoted Y_i, and temporary variables (i.e., WAM registers) by R_j. Instruction *moveregister* R_i,R_j replaces both *get_variable* R_j,R_i, and *put_value* R_i,R_j instructions.

An *inline* call is a call to an *inline* predicate, i.e., a deterministic builtin predicate which only changes a limited number of argument registers in a predictable way. An *out-of-line* call is a call to any other predicate (called *out-of-line* predicate). A *chunk* is a sequence of zero or more inline calls followed by an out-of-line call or by a ";" or a ")" indicating closing an or-branch. Arguments of inline calls may be taken from arbitrary registers, i.e., the first argument is not required to reside in R_1, the second in R_2, etc. We also use their classification of variables: A *void* variable is an variable which appears only once in a chunk. A *temporary* variable is a variable which does not survive the chunk; all other variables are *permanent*. A variable occurrence is a *top-level occurrence* if it is a argument of a predicate.

In the following, we restrict ourselves only to the first chunk of the clause (i.e., the head, the sequence of inline calls terminated by out-of-line call or ";"), because the treatment of any other chunk in the clause is just a special case of the former.

For the chunk, the following sets are defined:

- *Destination*(V), for each variable V of the chunk. It is the (possibly empty) set of registers R_i such that the i^{th} argument of the out-of-line call is a top-level occurrence of V.

- *HeadOccurrences*(V), for each variable V of the chunk. It is the number of occurrences of the variable V in the head of the clause.

These sets are global, i.e., they never change when we move through the clause. Several relevant local sets are defined as follows:

- *Free* is the set of free registers (unoccupied at the current position of clause traversal).

- *Vital* is the set of registers containing the only version of the value of variable V.

- *Assigned*(V), for each variable V, is the set of all registers containing V.

- *MTL* is the set of variables with dangerous multiple top-level occurrences in the head:

$$\begin{aligned} MTL \;=\; & \{V \mid \mathrm{card}(\mathit{Assigned}(V)) > 1 \wedge \\ & ((\mathit{Destination}(V) = \emptyset \wedge \mathit{UnPrep}(V) \neq \emptyset) \vee \\ & (\neg \exists R_j \in \mathit{Assigned}(V) : R_j \notin \bigcup_{W \neq V} \mathit{Destination}(W)))\} \end{aligned}$$

  (N.B.: Not all variables with the multiple top-level occurrences belong to this set).

- *Block*(R), for any register R, is the number of variables which cannot be moved to corresponding destination, when the argument in R is being processed (because their destination is blocked by another argument).

- *Half_block*(R), for any register R, is the number of variables which ought to be (directly or indirectly) moved to destinations occupied by some variable from *MTL*, when the argument in R is being processed.

- *Inline* is the list of all not yet processed inline calls. *Occ* is a set of unprepared argument variables associated with each inline call.
  A variable is *prepared*, after all head occurrences of this variable were processed in the course of unification.

- *UnPrep*(V), for each non-void variable V, is the set of head compound term arguments (subarguments) containing an occurrence of the variable V. *UnPrep*(V) $= \emptyset$ iff the variable V is prepared.

3 Complexity of Register Allocation

The problem in its full generality is just too intricate, hence we actually demonstrate NP-Completeness for a subproblem of the general one. Our result concerning register allocation problem is that it's at least NP-Complete. Demonstration of this is based on representing a particular register allocation task using a variable/register flow graph VRFG. This graph representation is as follows:

Given a clause with just one out-of-line call, no inline calls, no variable occurring multiply on top-level, and with no compound term with embedded compound term. We construct the directed graph VRFG with temporary registers as vertices: An arrow (directed edge) from vertex v_1 to v_2 indicates the need for moving a variable to its destination v_2 during the course of processing the argument placed in v_1. (Two simple properties of VRFG are easy to observe: Whenever v_1 hosts a variable, then at most one edge leads out of it. Whenever v_1 hosts a compound term, at most n edges lead out of it, with n being the arity of the term. The same applies to v_2.)

Using VRFG clause representation the version of register allocation problem (in its above-introduced simplified version) will be shown to be equivalent to "the minimum feedback set (FVS) problem" with NP-Completeness of this proved in [7].

FVS presents a problem of finding the minimal number of vertices to be eliminated from a directed graph in order the remaining graph be acyclic (without any loop).

Proof consists of several steps:

- First FVS is shown to be NP-Complete even when restricted to set $\mathcal{G}$ of directed graphs

$$\mathcal{G} = \{G = \langle V, E\rangle \mid \neg\exists \langle v_1, v_2\rangle = e \in E : out(v_1) > 1 \wedge in(v_2) > 1\}$$

  This restriction is motivated by a need of eliminating to represent the case of moving variables from inside a compound term into another place not in top-level occurrence; in which case a new register use does not contradict optimality.

- Then FVS for $G \in \mathcal{G}$ is reduced to the version of register allocation.

- Finally, the version of register allocation problem is reduced to the FVS for $G \in \mathcal{G}$.

For proving NP-Completeness of FVS for $\mathcal{G}$, a general directed graph $G = \langle V, E\rangle$ is transformed into a directed graph $\xi G = G' = \langle V', E'\rangle \in \mathcal{G}$ such that G' behaves in the same way as G with respect to FVS.

Definition of ξ :

1. $\forall v \in V : \exists v \in V'$

2. $\forall e = \langle v_1, v_2\rangle \in E, out(v_1) = 1 \vee in(v_2) = 1 : \exists e = \langle v_1, v_2\rangle \in E'$

3. $\forall e = \langle v_1, v_2\rangle \in E, out(v_1) > 1 \wedge in(v_2) > 1 : \exists v_e \in V', in(v_e) = out(v_e) = 1, \exists e_1 = \langle v_1, v_e\rangle \in E', \exists e_2 = \langle v_e, v_2\rangle \in E'$

4. $G' = \langle V', E'\rangle$, where V' and E' are the minimal sets satisfying the conditions 1–3.

G' is constructed from G by splitting some edges by inserting a new node on them (no loop in G can disappear and no new loop in G' can arise). Clearly, every solution of FVS for G is a solution of FVS for G'[1]. From the other side, every solution of FVS for G' can be transformed into solution of FVS for G in polynomial time.

Let M_0 be a set of vertices of G' such that $G' - M_0$ is acyclic, $\mathrm{card}(M_0) = N = \min\{\mathrm{card}(M) \mid G' - M \text{ is acyclic}\}$. When $M_0 \subset V$, M_0 is solution of FVS

[1]That means a minimal solution – the use of vertices from $V' - V$ cannot minimize the number of vertices removed, because such vertices lay each in at most one loop.

for G. Otherwise, start with $M_1 = M_0 \cap V$ and for each $v_0 \in M_0 \cap (V' - V)$ select a corresponding $v_1 : \langle v_0, v_1 \rangle \in E'$ (ξ guaranties existence of such a v_1) and insert it into M_1 (before this step M_1 cannot include v_1 because of optimality of M_0), then $\text{card}(M_1) = N$ and $G - M_1$ is acyclic (we have broken every loop).

M_1 is solution of FVS for G obtained from M_0 in $T(n)$ steps, where n is the number of vertices $v \in V$, with $in(v) > 1$.

Transformation ξ reduces every instance of FVS for general directed graph into a FVS for a graph from $\mathcal{G}$. Also every instance of FVS for $\mathcal{G}$ can be reduced to FVS for a general directed graph (due to inclusion). NP-Completeness for FVS for graphs from $\mathcal{G}$ follows from combination the two observations.

The second step starts with showing that every instance of FVS also represents a register allocation problem. From a directed graph $G = \langle V, E \rangle$, with $V = \{A_1, \ldots, A_n\}, E = \{X_1, \ldots, X_m\}$, the following defines the corresponding clause construction :

Head (p/n) and goal (q/n) of the clause will be terms with arity n.

The i$^{\text{th}}$ argument of the head (goal) depends on the number of the outgoing (incoming) edges from (to) vertex A_i in the following way :

- if there is no edge outgoing from (incoming to) vertex A_i, then the i$^{\text{th}}$ argument of the head (goal) is atom p_i (q_i)
- if there is only one edge X_j outgoing from (incoming to) vertex A_i, then the i$^{\text{th}}$ argument of the head (goal) is the variable X_j
- if there are k edges $(X_{j_1}, \ldots, X_{j_k}; 1 < k \leq m)$ outgoing from (incoming to) vertex A_i, then the i$^{\text{th}}$ argument of the head (goal) is a compound term $p_i/k : p_i(X_{j_1}, \ldots, X_{j_k})$ $(q_i/k : q_i(X_{j_1}, \ldots, X_{j_k}))$

Clearly, the clause we have constructed satisfies the restriction defined at the beginning of this section. Time needed for its construction is $T(n + m)$, where $n = \text{card}(V), m = \text{card}(E)$. We have provided the clause construction for arbitrary graph G, so in particular this can be done for every member of $\mathcal{G}$. VRFG corresponding to the clause constructed is directly G itself.

Now we will show that a solution of FVS for $G \in \mathcal{G}$ (VRFG) is itself a solution of the version of the register allocation problem, and vice versa.

- If VRFG $\in \mathcal{G}$ does not contain any loop, it is possible to generate the code for the associated clause without using any new register. The generation starts in some vertex from which every edge goes to a leaf (existence of such a vertex in any acyclic graph is guaranteed). After that we remove all such leaf-terminated edges from VRFG. New VRFG$'$ is also acyclic, and the process of generation continues until VRFG$'$ contains no edge at all.

- If VRFG $\in \mathcal{G}$ does contain a loop, then any order of processing of the vertices of the loop causes a register conflict[2], i.e., we must use a new register, save a value of the argument in it (an additional *moveregister*), generate code for remaining vertices of the loop, and then process saved argument. It is not true that we need an extra register for each loop but the minimal number of extra registers[3] needed is equal to the minimal number of vertices which must be eliminated from the VRFG so that the remaining graph is acyclic.

The two cases above show the equivalence of the solutions in trivial cases (i.e., the minimal number of vertices/registers is zero). In a nontrivial case, another transformation of graphs which breaks loops in the same place as removing of vertices must be used:

The transformation $\mathcal{T}$ replaces a vertex v by two vertices v', v'' in such a way that all edges incoming to the vertex v become incoming to v' instead, and all edges outgoing from v become outgoing from v''.

- Denote VRFG$_{\mathcal{V}}$ the graph formed from VRFG by removing all vertices in $\mathcal{V}$ (VRFG$-\mathcal{V}$) and VRFG$^{\mathcal{T}}_{\mathcal{V}}$ the graph formed from VRFG by transformation $\mathcal{T}$ applied on each vertex $v \in \mathcal{V}$. VRFG$_{\mathcal{V}}$ and VRFG$^{\mathcal{T}}_{\mathcal{V}}$ have the same number of loops.

- Let VRFG$_{\mathcal{V}}$ be acyclic, with $\mathcal{V}$ minimal, $N = \mathrm{card}(\mathcal{V})$, and $N_R = \mathrm{card}(\mathcal{V}')$ be the minimal number of registers needed to solve the register conflicts in VRFG.

  1. As far as VRFG$^{\mathcal{T}}_{\mathcal{V}}$ is acyclic, it is possible to generate code for it without any new registers. VRFG$^{\mathcal{T}}_{\mathcal{V}}$ was formed from VRFG by N transformations $\mathcal{T}$, hence $N \geq N_R$.
  2. VRFG$^{\mathcal{T}}_{\mathcal{V}'}$ corresponds to the situation after moving values of N_R registers from $\mathcal{V}'$ to new registers, i.e., VRFG$^{\mathcal{T}}_{\mathcal{V}'}$ is conflict free. VRFG$_{\mathcal{V}'}$ was formed from VRFG by removing N_R vertices, hence $N_R \geq N$.

  From the inequalities above it follows $N = N_R$, i.e., the minimal number of registers needed is equal to the minimal number of vertices which must be removed from VRFG so that the resulting graph is acyclic.

What remains to be proved is that every register allocation task within the version introduced at the beginning of this section can be reduced to

[2] This need not be true for cyclic $G \notin \mathcal{G}$, if a loop contains an edge representing the move of a variable from a compound term into another compound term. In this case, we can solve the conflict by starting from the vertex the edge outgoes from, and using new register preserves optimality. Moreover, for $G \in \mathcal{G}$ there is one exception – sequence of moves over top-levels starting in compound term and ending in the same place on top-level, which is solved automatically.

[3] More precisely: the number of extra *moveregister* instructions, because in certain cases one register can be used several times for different arguments.

FVS. For every clause satisfying our restriction we construct VRFG, from VRFG we remove all edges representing value transfers between compound terms and one-edge loops (we are allowed to do so, because these edges have no effect to the order of processing and its optimality). As a sequent we obtain VRFG$' \in \mathcal{G}$. Solving of FVS for VRFG$'$ is equal to solving of register allocation problem of the associated clause with VRFG$'$ as shown earlier. Note that the associated clause may differ from the original clause for which VRFG$'$ was constructed, but both the clauses behave in the same way with respect to the register allocation (i.e., we need the same number of extra registers for processing such clauses, e.g., $p_1(s(X),t(Y))$:–$q_1(Y,X)$ and $p_2(X,Y)$:–$q_2(Y,X)$).

4 Optimal Allocation Strategy

In the previous section we have shown that the register allocation problem in WAM is at least NP-Complete. Nevertheless, there are many interesting cases for which optimal computationally tractable strategies can be found.

As was pointed out in [8], the parameter passing in WAM-based Prolog implementations is side effect free, and the order in which the individual head arguments are to be processed may be adapted in order to minimize the number of register moves (i.e., the number of instructions generated by the compiler). Moreover, inline call processing may be interspersed with argument processing. Correctness of this reorganization is guaranteed if no inline call is to be processed before all variables in its arguments are prepared, and the order of inline calls is preserved[4]. Head argument and inline call processing may also be interspersed with the preparation of the out-of-line arguments. In general, this is possible only if all the variables of a compound term argument are prepared and all non-test inline calls[5] using this variable are processed[6] as well.

In the sequel we will design and evaluate a polynomial strategy, optimal for a restricted class of clauses. The rest of this section describes this strategy.

First, global sets *Destination* and *HeadOccurrences* are generated, together with the initial values for all local sets. Then local sets are passed from program point to program point, updated, and used for code generating.

The contents of the local sets *Block* and *Half_block* is determined by the following:

[4]This is not exactly true, because the sufficient condition is the preservation of the order of mutually dependent inline calls, but the problem of finding mutually independent sets of inline calls is out of the scope of this paper.

[5]A test inline call is a builtin predicate which does not change the value of its arguments, i.e., *var/1*, ==, \=, \==.

[6]Again, this is more strict then actually necessary.

- For any constant head argument, the *Block* value is zero (this argument may be processed immediately).

- For a variable V in a register R, *Block*(R) is zero if *Destination*(V)= $\emptyset$ or for some register $R_j \in$ *Destination*(V) $\wedge$ $R_j \in$ *Free*, or for some register $R_k \in$ *Destination*(V) there exists a sequence of the *moveregister* instructions over non-blocked variables which frees register R_k. For any permanent variable Y *Block*(Y) = *Half_block*(Y) = 0.

- For a compound term in a register *R*, *Block*(R) is the sum of *Block* values of all first level argument variables.

Half_block values are computed only for variables with *Block*(V) = 0 and for compound terms.

- *Half_block*(R) for a variable V in a register R equals to one if R_j from the *Destination*(V) is in the *MTL*, or if the sequence of the *moveregister* instructions found ends in such a register.

- *Half_block* value of the compound term equals to the sum of *Half_block* values of its first level variable arguments.

The search for the optimal *moveregister* instruction sequence is performed by a simple modification of the *Release* function from [8]. The computation of *Block* and *Half_block* values for any compound term must be done as a unit, i.e., the order of arguments must be preserved (this is properly settled down in the modified *Release* function which when processing the n^{th} argument of a compound term, it employs the results of processing all the previous $n-1$ term arguments).

When the *Block* (and *Half_block*) values for all the registers containing head arguments are computed, the following takes place:

- First, all arguments with *Block*(R) = *Half_block*(R) = 0 are processed (in the left to right order). Not only all *known* (for definition of *known* cf. [8]) arguments are processed in this way, but all unblocked (variable or compound) arguments are processed as well. The processing order also complies with suggestions from [9]. Only the first level arguments of compound terms are processed, their compound subarguments are moved to registers from *Free*, and their processing is delayed (till corresponding *Block*, *Half_block*, and other values and sets are known).

- Second, all variables with multiple top-level occurrences are processed (*get_value* instructions are generated), thus facilitating early detection of failure.

As the unblocked argument processing may also free other arguments, the previous two steps are repeated until no more arguments may be processed in this way (or when no more arguments remain at all).

When this is completed, beginning of the *Inline* list is inspected. If this item has empty *Occ* set, the corresponding inline call is processed and removed from the *Inline* list. The inspection, and code generation is repeated until the *Inline* list is empty or the beginning item has non-empty *Occ* set. This eager strategy of inline call processing facilitates early detection of failure, and it is used through the whole process of code generation.

So far, the optimality of generated code is guaranteed (i.e., all variables are put into their destinations, new registers are used only for new variables and new compound terms, and the minimal number of instructions is generated). In the following, the arguments with $Block(\_) = 0$ and $Half\_block(\_) \neq 0$ are processed from left to right, but, unfortunately, this may not be the optimal order. The *Block* and *Half_block* values must be updated after processing of each argument is completed: destruction of some copies of the *MTL* variables may block processing of some other arguments.

When several arguments with $Block(\_) \neq 0$ remain, there is at least one register conflict unsolvable without the use of a new register (and generation of *moveregister* instruction). We know from the previous section that it is "hard" to find the optimal order in which arguments must be treated. However, several useful and not too costly heuristics may be used, and many common cases may be solved optimally or almost optimally.

The main idea behind the heuristics is the *transfer* of a whole argument to a new register. Clearly, it is cheaper to move the argument with $Block(\_) > 1$ to a new register than to process this argument, because in the former case we release the argument register at the cost of one free register whereas in the later case the cost is $Block(\_)$ registers.

Also, the processing of some out-of-line call arguments interspersed with the head unification may help a lot. A *Vital* copy of some variable may be "consumed" by a compound term in some argument of the out-of-line call (i.e., used as the subargument of this compound term), thus unblocking some head argument.

Several useful notions may be derived from the VRFG representation:

- Two blocked arguments are *independent* if they belong to disjoint loops in VRFG (i.e., they have no common vertex) and there does not exist any loop with vertices common to both the loops.

- Two blocked arguments are *fully independent* if they belong to disjoint loops in VRFG and there does not exist any path between these two loops (e.g., the VRFG is disconnected and the loops belong to different parts of the VRFG).

- We define a relation $\sqsubset$ between two registers R_i and R_j containing *independent* blocked arguments. $R_i \sqsubset R_j$ iff there exists a path from some vertex of the R_j argument loop to some vertex of the R_i argument loop. The relation $\sqsubset$ is a total order on independent but not fully independent blocked arguments.

- Mutually dependent arguments form a *cluster*. Only clusters must be treated as units, and only for them the register allocation problem is NP-Complete. The $\sqsubset$ relation is also applicable on clusters, and individual clusters must be processed in the order posed by this relation.

- The processing order of fully independent arguments is immaterial.

Although loop independence computation is not the cheapest conceivable operation, it may be worth the effort, at least for some highly optimizing Prolog compiler. It's a well-known trivial fact that it's painful to decide between slow compiler producing fast code and the opposite ...

The following heuristics may be proposed for the treatment of blocked arguments:

1. If there exists a variable V with $UnAPrep = \emptyset$[7] then try to process all out-of-line call arguments containing this variable (it may be advisable to construct for each variable a set of all out-of-line arguments containing the variable, and also to construct for each out-of-line argument a set of variables it contains: even the check-on-demand is not expensive). If some out-of-line argument "consumes" more then one blocking variable, it may be useful to process this argument even if its destination is not yet free (release of two or more registers at the cost of one free register).

2. If there exists a variable V in a register R such that the $UnPrep(V)$ contains just one compound term argument and there exists some other variable W with $Destination(W) = \{R\}$, then try to process the $UnPrep(V)$ compound term if its $Block(\_) = 1$. The rationale behind this heuristic is the chance to unblock some variable or compound term blocked by the vital copy of V.

3. Find the vertex R with the highest degree in a cluster and transfer it to a new register. If there is more then one such vertex, choose the one with the highest input degree. The output degree of a vertex in the VRFG is the *Block* value, and the input degree is the number of variables – arguments of a term in R – whose $Destination = \emptyset$. The release of register R enhances the chance to unblock some variable or compound term blocked by it.

The heuristics should be applied in the indicated order.

The proposed heuristics either deal with the sets already used in the previous parts of our register allocation strategy, or assure that the maintenance cost of the new introduced sets is low. The heuristics are able to produce optimal code in many cases, and we claim the code generated by them to be nearly optimal under any circumstances.

[7] *UnAPrep*, for each non-void variable V, is the union of $UnPrep(V)$ with the set of all non-test inline calls using V.

5 Evaluation

5.1 Comparison with Previous Work

We compare our method with the "adaptable unification order" of Janssens et al. [8] and strategy in [17], which are the best register allocation algorithms published so far.

The "adaptable" method finds the optimal order for the treatment of variables as head arguments, but it may fail when compound term arguments are involved. Our method uses extended *Release* function, and finds the same unification order for variables as arguments as the "adaptable" method. Moreover, the *Block* and *Half_block* sets allow to find the optimal order even for a subset of compound terms as arguments. In [8], clause *head(f(T),g(T)) :- goal(a,T)* is used as an example of incompleteness of their algorithm:

"adaptable" strategy	new strategy
get_structure f/1,R_1	*get_structure g/1,*R_2
unify_variable R_3	*unify_variable* R_2
get_structure g/1,R_2	*get_structure f/1,*R_1
unify_value R_3	*unify_value* R_2
moveregister R_3,R_2	*put_constant* a,R_1
put_constant a,R_1	

This result is achieved because $Block(R_1) = 1$ and $Block(R_2) = 0$, and thus the instruction generation starts from the second argument.

More complicated cases are also processed optimally. The code generated for *head(b(X,Z),Z,Y,a(X)) :- goal(c,Y,X,Z)* is:

"adaptable" strategy	new strategy
get_structure b/2,R_1	*get_structure* a*/1,*R_4
unify_variable R_5	*moveregister* R_2,R_4
unify_local_value R_2	*moveregister* R_3,R_2
get_structure a*/1,*R_4	*unify_variable* R_3
unify_value R_5	*get_structure b/2,*R_1
moveregister R_2,R_4	*unify_value* R_3
moveregister R_3,R_2	*unify_local_value* R_4
moveregister R_5,R_3	*put_constant c,*R_1
put_constant c,R_1	

Again, $Block(R_1) = Block(R_2) = Block(R_3) = 1$ and $Block(R_4) = 0$ (because there exists a sequence of the *moveregister* instructions which frees the R_4 register).

A similar pattern may be found in the following example, where the interspersing of argument unification and inline calls is shown. The code generated for *head(g(Y),s(X,h(Y)),f(X)) :- var(X), var(Y), goal(a,Y,X)* is:

"adaptable" strategy	new strategy
get_structure g/1,R_1	*get_structure f/1,*R_3
unify_variable R_4	*unify_variable* R_3
get_structure s/2,R_2	*get_structure s/2,*R_2
unify_variable R_5	*unify_value* R_3
unify_variable R_6	*unify_value* R_2
get_structure h/1,R_6	*call_builtin var/1,*R_3
unify_value R_4	*get_structure h/1,*R_2
get_structure f/1,R_3	*unify_variable* R_2
unify_value R_5	*get_structure g/1,*R_1
call_builtin var/1,R_5	*unify_value* R_2
call_builtin var/1,R_4	*call_builtin var/1,*R_2
moveregister R_4,R_2	*put_constant* a,R_1
moveregister R_5,R_3	
put_constant a,R_1	

The "adaptable" method not only generates two more instructions (and uses three more registers), but if the predicate is called with X instantiated, the failure on *var(X)* is detected later.

An example of global register conflict solvable by one of heuristics proposed follows. The code generated for the clause *head(a(X,Y,Z),b(Y),c(D), A,d(Y,A,D),e(Q)) :- goal(A,X,Y,D,Q,Z)* is:

"adaptable" strategy	new strategy
get_structure a/3,R_1	*moveregister* R_1,R_7
unify_variable R_7	*get_structure c/1,*R_3
unify_variable R_8	*moveregister* R_4,R_1
unify_variable R_9	*unify_variable* R_4
get_structure b/1,R_2	*get_structure b/1,*R_2
unify_value R_8	*unify_variable* R_3
get_structure c/1,R_3	*get_structure d/3,*R_5
unify_variable R_{10}	*unify_value* R_3
get_structure d/3,R_5	*unify_local_value* R_1
unify_value R_8	*unify_value* R_4
unify_local_value R_4	*get_structure e/1,*R_6
unify_value R_{10}	*unify_variable* R_5
get_structure e/1,R_6	*get_structure a/3,*R_7
unify_variable R_5	*unify_value* R_2
moveregister R_4,R_1	*unify_value* R_3
moveregister R_7,R_2	*unify_variable* R_6
moveregister R_8,R_3	
moveregister R_{10},R_4	
moveregister R_9,R_6	

Here again our strategy outperforms the "adaptable" method and generates less instructions (in fact, the generated code is optimal). The argument in the first register has the highest degree (3 outgoing edges plus 2 incoming ones) and thus is selected for transfer. Then $Block(R_4) = 0$, after its treatment $Block(R_3) = 0$, then $Block(R_2) = 0$, and so on until the transferred argument in R_1 may also be processed.

Also when comparing to [17], our approach gives better results. The code generated for clause *head(f(X),Y,g(Y)) :- goal(a,X,b)* is:

strategy in [17]	new strategy
get_structure f/1,R_1	*get_structure g/1,*R_3
unify_variable R_4	*unify_value* R_2
get_structure g/1,R_3	*get_structure f/1,*R_1
unify_value R_2	*unify_variable* R_2
moveregister R_4,R_2	*put_constant a,*R_1
put_constant a,R_1	*put_constant b,*R_3
put_constant b,R_3	

Better code is generated due finding $Block(R_3) = 0$, while $Block(R_1) \neq 0$.

5.2 Future Work

The presented register allocation method is optimal only for clauses where no global register conflict exists and the *MTL* set is empty. In this case the number of data transfers between registers is minimized, and the smallest (and fastest) code is generated. However, the optimality may be lost if the clause contains multiple head top-level occurrences of variables. Also, as the general blocking problem has been shown to be at least NP-Complete, there is an open question of strong heuristics for the generation of optimal code in restricted cases.

A new method for the compilation of compound terms in Prolog was recently proposed [12]. This method allows a depth first traversal of the compound term, instead of the breadth first traversal used in WAM (and also in our method). The preliminary analysis has shown that there are cases when the Meier's method of compound term compilation may lead to fewer instructions generated then our allocation strategy, but the opposite also occurs. The practical implication for the register allocation strategy presents an open research issue.

The eager processing of inline calls in the chunk enhances the possibility of discovering failure as soon as possible, but even more information may be taken from the inline predicates. For example, full treatment of =/2 builtin predicate may lead to a better code generated. The full treatment of inline predicates must be accompanied with the computation of mutually independent sets of these, and this will also enable earlier failure detection compared to our strategy.

The proposed register allocation strategy is being implemented. An early prototype system written in Prolog is used for testing the performance improvements and possible new heuristics. The full implementation will replace the register allocation module of the MUB-Prolog WAM based compiler, developed at the University of Brno [11].

6 Conclusion

The unification strategy proposed is an almost optimal method for register allocation in the WAM. It is based on cutting clauses into chunks, computing the *blocking* numbers for individual registers, and unification order control relying on the blocking numbers computed. The method proposed is optimal only for a subclass of all possible cases, but this subclass is much larger than subclasses of the recently published methods [8, 9, 17].

Besides of the good news of obtaining a register allocation strategy superior to those published so far, the bad news is that any optimal method for register allocation in WAM must be at least NP-Complete. Even in the light of this, several heuristics leading to nearly optimal code generation at low cost have been proposed.

Acknowledgements

This research has been supported by “IBM Academic Initiative (SFR 1990)”.

References

[1] Aït-Kaci, H.: The WAM: A (Real) Tutorial, *Tech. report 5*, DEC, Paris Research Laboratory, January 1990.

[2] Carlsson, M.: Freeze, Indexing, and Other Implementation Issues in the WAM, *Proc. 4th Int. Conf. Logic Programming*, pp.40–58, MIT Press, Cambridge, MA, 1987.

[3] Debray, S. K.: Register Allocation in a Prolog Machine, *Symposium on Logic Programming*, pp.267–275, IEEE, 1986.

[4] Debray, S. K. and Warren, D. S.: Automatic Mode Inference for Logic Programs, *J. Logic Programming*, 5:207–229, 1988.

[5] Debray, S. K.: Unfold/Fold Transformations and Loop Optimization of Logic Programs, *Proc. ACM SIGPLAN'88*, pp.297–307, ACM, New York, 1988 (*SIGPLAN Not.* 23, 7).

[6] Debray, S. K.: Static Inference of Modes and Data Dependencies in Logic Programs, *ACM Trans. Prog. Lang. Syst.*, 11:418–450, 1989.

[7] Even, S.: *Graph Algorithms*, Computer Science Press, Rockville, Maryland, 1979, pp.223–224.

[8] Janssens, G., Demoen, B. and Martën, A.: Improving the Register Allocation in WAM by Reordering Unification, *Proc. 5th Int. Conf. Symp. Logic Programming*, pp.1388–1402, MIT Press, Cambridge, MA, 1988.

[9] Kurosawa, S. A. K. and Kiriyama, K.: A new Optimization Technique for a Prolog Compiler, *IEEE COMPCON'86*, San Francisco, CA, pp.241–245, Computer Society Press, Rockville, Maryland, 1986.

[10] Lindholm, T. and O'Keefe, R. A.: Efficient Implementation of a Defensible Semantics for Dynamic Prolog Code, *Proc. 4th Int. Conf. Logic Programming*, pp.21–39, MIT Press, Cambridge, MA, 1987.

[11] Matyska, L. and Toman, D.: Prolog and WAM – A Case Study of an Implementation, *Proc. LOP'88*, pp.65–88, Ruprechtov, 1988 (Available also as the *Tech. Report*, ICS MU 1988).

[12] Meier, M.: Compilation of Compound Terms in Prolog, *Proc. NACLP'90*, MIT Press, Cambridge, MA, 1990.

[13] Umrigar, Z. D.: Finding Advantageous Orders for Argument Unification for the Prolog WAM, *Proc. NACLP'90*, pp.80–96, MIT Press, Cambridge, MA, 1990.

[14] Warren, D. H. D.: An Abstract Prolog Instruction Set, *Tech. Report 309*, SRI International, AI Center, Palo Alto, 1983.

[15] Warren, R., Hermenegildo, M. and Debray, S. K.: On the Practicality of Global Flow Analysis of Logic Programs, *Proc. Int. Conf. Symp. on Logic Programming*, pp.684–699, Seattle, 1988.

[16] Xu, J. and Warren, D. S.: A Type Inference System for Prolog, *Proc. 5th Int. Conf. Symp. Logic Programming*, pp.604–619, MIT Press, Cambridge, MA, 1988.

[17] Hu Ziang, Xiang Rong Ci: An Improved Register Allocation Algorithm for the WAM Model, *IEEE Proc. Tools for AI'90*, pp.651–657, 1990.