Register Allocation in a Prolog Machine

Saumya K. Debray
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794

Abstract: We consider the Prolog Engine described by D. H. D. Warren. An interesting feature of this machine is its parameter passing mechanism, which uses register k to pass the $k^{th.}$ parameter to a procedure. Warren distinguishes between *temporary* variables, which can be kept in registers in such a machine, and *permanent* variables, which must be allocated in memory. The issue in register allocation here is one of minimizing data movement, and is somewhat different from that of minimizing the number of *load*s and *store*s, as in a conventional machine. We describe three register allocation algorithms for such a machine. These strategies rely on a high-level analysis of the source program to compute information which is then used for register allocation during code generation. The algorithms are simple yet quite efficient, and produce code of good quality.

This work was supported in part by the National Science Foundation under grant number DCR-8407688.

1. Introduction

With the growing interest in logic programming languages, especially Prolog, the issues of specific architectures supporting such languages, and algorithms relating to such architectures, become especially relevant. Warren, addressing the former issue, describes an abstract Prolog machine and its instruction set [4]. This design has formed the basis of a number of Prolog implementations, including ones at Quintus Systems, Argonne National Labs, UC Berkeley, Syracuse University and SUNY at Stony Brook. An interesting feature of this machine is its parameter passing convention: register k is used to pass the $k^{th.}$ parameter of a procedure call. Warren distinguishes between ‘‘temporary’’ and ‘‘permanent’’ variables, and notes that temporary variables can be kept in registers. He points out that by properly allocating registers to temporary variables, the code generated for a program can be improved substantially, with a significant reduction in the number of instructions that have to be executed, and a concomitant increase in speed.

Warren sketches a fairly low-level approach to this problem in [4], which analyses the abstract machine instructions generated to effect the optimization. The Berkeley PLM compiler uses a backtracking-driven algorithm for register allocation [3]; this, however, can be quite slow. This paper describes an alternative approach, which relies on a high level analysis of the source program to produce annotations to the program. These annotations are then used by the code generator while allocating registers to temporary variables. These algorithms differ from traditional register allocation algorithms (see, for example, [1, 2]) in that the analysis of the source program produces information regarding which registers should be allocated to a variable, rather than which variables should be allocated to registers.

The remainder of this paper is organized as follows: Section 2 discusses some preliminary notions. Section 3 develops the basic ideas behind our algorithms, and Section 4 goes on to describe three register allocation strategies. Section 5 outlines an extension of these ideas to global register allocation. Section 6 concludes with a summary.

2. Preliminaries

This section considers some preliminary issues. The syntax of Prolog is described briefly, followed by a discussion of temporary variables. The section concludes with a brief outline of the instruction set of the Warren Prolog Engine.

2.1. Syntax

A Prolog term is either a variable, or a constant, or a compound term $f(t_1, ..., t_n)$ where f is an n-ary function symbol and $t_1, ..., t_n$ are terms. A literal is either an atom $p(t_1, ..., t_n)$ where p

is an *n*-ary predicate symbol and t_1, ..., t_n are terms, or the negation of an atom. A clause is a finite sequence of literals. A clause is said to be Horn if it has at most one positive literal. A Horn clause with exactly one positive literal is referred to as a *definite clause*. A predicate definition consists of a sequence of definite clauses. A program consists of a set of predicate definitions.

In this paper, we will adhere to the syntax of DEC-10 Prolog and write a definite clause as

$$p :- q_1, q_2, \ldots, q_n.$$

which can be read declaratively as ''q_1 *and* q_2 *and* . . . *and* q_n *implies* p''. The clause can also be interpreted procedurally as meaning that the body of procedure p is defined as the sequence of procedure calls q_1, q_2, . . ., q_n. The q_i's will be referred to as *goals*. Following DEC-10 syntax, variables names will begin with upper case letters, all other names with lower case letters. Given a literal $p(t_1, \ldots, t_n)$, if t_i is a variable X for some i, then X will be said to *occur at the top level* at argument position i in that literal.

2.2. Temporary Variables

A temporary variable in a Prolog clause, as defined by Warren, is a variable which has its first occurrence in the head of the clause, in a compound term or in the last goal, and which does not occur in more than one goal. The head is counted as part of the first goal. Any variable which is not temporary is referred to as *permanent*. The point behind this distinction is that while permanent variables must be allocated space in memory, in the activation record of the procedure to which it belongs, temporary variables may be stored in registers. With the proper allocation of registers to temporary variables, the efficiency of programs can be improved significantly.

The reasons for this definition of temporary variables are the following: if the first occurrence of a variable is in the head or in a compound term, then it is guaranteed to appear deeper in the local stack or on the heap, so that pointers can safely be set to it, even if the variable is kept in a register; if its first occurrence is in the last goal of a clause, then it must be allocated on the heap to enable *last goal optimization* (a generalization of tail recursion optimization) to be performed, but this turns out to be equivalent to treating the variable as temporary. A temporary variable is not permitted to occur in more than one goal because no assumption is made about the register usage in other procedures. Therefore, if a variable appears in more than one goal, it cannot be kept in a register, but has to be saved in the activation record for the predicate, i.e. made *permanent*.

We extend this definition of temporary variables slightly. To this end, we define the notions of *in-line predicates* and *chunks*:

Definition: An *in-line* predicate is one which can be executed in-line by a sequence of Prolog machine instructions, without having to branch to a procedure.

Typically, in-line predicates are those that perform primitive computations, e.g. arithmetic and relational computations, unification, testing to see whether or not a variable has been bound to a non-variable term, etc.

Definition: A *chunk* is a sequence of zero or more in-line predicate followed by a predicate which is not in-line.

Each chunk in a clause represents the longest computation at that point which can be performed with only one procedure call, at the end. Since the code for in-line predicates can be executed without making any procedure calls, the constraints imposed by the parameter passing convention are absent. For each in-line predicate in a chunk, we can determine which registers are used; however, no assumptions are made regarding register usage in predicates that are not in-line. Thus, register usage can easily be determined all through the chunk, upto its last goal. This allows us to extend the definition of temporary variables to the following:

Definition: A variable is temporary if its first occurrence is in the head of a clause or in a compound term or in the last chunk, and which does not occur in more than one chunk, with the head being counted as part of the first chunk.

2.3. The Instruction Set

The instruction set we use is a superset of that originally proposed by Warren. There are five basic groups of instructions: *get* instructions, *put* instructions, *unify* instructions, *procedural* instructions and *indexing* instructions. In addition, there are other instructions for arithmetic operations, language features such as cut, etc., which do not pertain directly to the subject of this paper and will not be considered further. While a detailed description of the instruction set is beyond the scope of this paper, we give a brief overview of the basic instruction groups. Details may be found in [4-6].

The *get* instructions correspond to the arguments at the head of a clause and are responsible for matching against the procedure's arguments. The *put* instructions correspond to the arguments of a goal in the body of a clause and are responsible for loading the arguments into the appropriate registers. The *unify* instructions correspond to the arguments of a compound term. In each group, there are instructions for constants (e.g. *get_constant*, *unify_constant*), compound terms (e.g. *put_structure*), and the first and subsequent occurrences of variables. The instructions for variables are parameterized by type, so that there is an instruction for temporary and one for permanent variables. Thus, for the first occurrence of temporary and permanent variables in the head we have, respectively, the instructions *get_temp_variable* and *get_perm_variable*. The *get_temp_variable* instruction has the same functionality as the more conventional *movreg* instruction for register to register moves, so we will use *movreg* in its place.

The *procedural* instructions correspond to the predicates that form the head and goals of the clause, and are responsible for control transfer and environment allocation associated with procedure calling. These include *call* (for procedure calls), *execute* (similar to call, except that it is used for the last goal in a clause and does last goal optimization) and *proceed* (analogous to *return*). The *indexing* instructions are responsible for linking together the code for different clauses of a predicate.

2.4. Soundness Criteria for Register Allocation Algorithms

Any admissible register allocation algorithm must be *sound*. Since temporary variables are kept in registers, our soundness criteria are that, at all points between the first and last occurrences of any temporary variable T, (i) T must be present in some register R, i.e. its value must not be destroyed; and (ii) the fact that T is in register R must be known, i.e. it must not be lost.[1]

This is perhaps stronger than necessary, since correctness of compilation requires only that a temporary variable be present in a register when it is needed; it may be stored in the activation record of the procedure, like a permanent variable, at any other point, and the register released. This would also be closer to the treatment of variables in implementations of traditional programming languages. However, this violates the spirit of distinguishing between temporary and permanent variables, so we do not consider it.

Any variable appearing in more than one chunk is permanent by definition. Therefore, to establish the soundness of a register allocation algorithm, it suffices to establish its soundness for each chunk.

---

[1] If the number of registers is insufficient, the temporary variable will have to be spilled, i.e. made permanent. We assume this throughout the paper, and do not state it explicitly.

3. Developing Register Allocation Algorithms

3.1. A Naive Register Allocation Strategy

The simplest register allocation strategy involves no analysis of the code at all. A list *INUSE* of registers that are in use is maintained. If a register is not in *INUSE*, it will be said to be *free*. When a register has to be allocated to a new temporary variable, an arbitrary free register is chosen. When a register becomes free (e.g. after the last occurrence of the temporary variable it had been allocated to), it is removed from *INUSE*. While the algorithm is simple, the code produced can be far from optimal.

3.2. Improving the Algorithm

It is possible, in principle, to perform register allocation using the naive strategy and subsequently improve the code generated by making a separate pass over it. We propose a simpler solution which involves analysing variable occurrence information in the symbol table. For each occurrence of a variable, we need to know (*i*) its type, i.e. temporary or permanent; (*ii*) which argument in the goal it occurs in; and (*iii*) whether the variable occurs at the top level or within a compound term. Since this information is already needed for subsequent code generation, no extra overhead is incurred.

The key observation in our register allocation algorithms is that if a temporary variable V occurs as the $N^{th.}$ top-level argument of the last goal in a chunk, then, because of the parameter passing convention, V will ultimately have to be moved to register N. It is therefore reasonable to try to allocate register N to V in the first place. Additionally, for the first chunk of a clause the situation is mirrored by top-level occurrences of variables in the head: if a temporary variable V occurs as the $N^{th.}$ top-level argument in the head of the clause, then it will be in register N at entry to the clause. It is therefore reasonable to try to allocate register N to it, so as to avoid having to move it. These ideas are incorporated into the register allocation scheme by associating a set, called the *USE* set, with each temporary variable in a clause.

Definition: The set *USE*(T) associated with a temporary variable T is the set of argument positions corresponding to top-level occurrences of T in the last goal of that chunk, and in the head of the clause if the chunk is the first in the clause.

Example: Consider the clause

p(X,f(X),Y) :– X = [a|_], q(Z,Y,X), r(Z).

The temporary variables in this clause are X and Y. The top-level occurrences of X are as argument 1 in the head and as argument 3 in the call to *q* (the last goal in the first chunk); thus, *USE*(X) = {1,3}. Similarly, *USE*(Y) = {2,3}.

Our first ideas about a register allocation algorithm are as follows: we proceed as before, except that when a temporary variable *X* has to have a register allocated to it, a free register *r* in *USE*(X) is allocated to *X* whenever possible. We henceforth adopt the convention that registers are allocated upwards from the low numbers, i.e. register 1 is considered for allocation first, then register 2, then 3, and so on.

The code produced by this modified algorithm is still not entirely satisfactory, however. What may happen is that a temporary variable may be allocated a register it does not really need, but which is needed by some other variable. In order to prevent this from happening, we associate another set, the *NOUSE* set, with each temporary variable.

Definition: The set *NOUSE*(*T*) associated with a temporary variable *T* in a chunk is the set of argument positions *m* corresponding to top-level occurrences of temporary variables *U*, $U \neq T$, in the last goal of the chunk, such that *m $nomem* *USE*(*T*).

The idea is that members of the *NOUSE* set of a variable are those registers that it does not need, but which other temporary variables might need. As before, we consider only the last goal in the chunk because we are not bound by any register usage convention for the in-line predicates before it. Thus, for a clause

p(X,f(X),Y) :– X = [a | W], q(W,Y,X), r(Z).

we have *NOUSE*(X) = {2}, since the temporary variable Y occurs in argument position 2 in the call to *q/3*, and 2 is not in *USE*(X). Note that even though the temporary variable W occurs in the first argument position of this literal, 1 is not in *NOUSE*(X) because it is already in *USE*(X). Similarly, *NOUSE*(Y) = {1}.

The register allocation algorithm is modified as follows: when allocating a register to a temporary variable, if it cannot be allocated a register in its *USE* set, then it is allocated a free register not in its *NOUSE* set.

Example: Consider the clauses

append([ ], L, L).
append([H|L1], L2, [H|L3]) :– append(L1, L2, L3).

In the first clause, *USE*(L) = {2,3} and *NOUSE*(L) = ∅. L can be allocated either register 2 or register 3; in both cases, the code generated is

getnil 1

```
get_temp_value 2, 3
proceed
```

In the second clause, we have *USE*(H) = ∅, *NOUSE*(H) = {1,2,3}; *USE*(L1) = {1}, *NOUSE*(L1) = {2,3}; *USE*(L2) = {2}, *NOUSE*(L2) = {1,3}; *USE*(L3) = {3}, *NOUSE*(L3) = {1,2}. From this, H is allocated register 4, L1 is allocated register 1, L2 is allocated 2, and L3 is allocated register 3. The code generated is

```
get_list 1
unify_temp_var 4
unify_temp_var 1
get_list 3
unify_temp_value 4
unify_temp_var 3
execute append/3
```

3.3. Handling Conflicts

We have tacitly assumed, thus far, that a register can be dedicated to a temporary variable throughout the lifetime of that variable. However, owing to the parameter passing convention of the Prolog machine, this does not suffice to avoid conflicts. The problem is that the allocation strategy we have developed so far does not take into account the fact that a conflict may arise because a register is needed to pass a parameter of a call. What can happen is that a temporary variable may be allocated register *n*, which is then found to be needed for the $n^{th.}$ parameter of a call. There are two basic ways of dealing with such conflicts: *avoidance* and *resolution*.

In *conflict avoidance*, all possible sources of conflict are computed beforehand, and registers allocated based on this information so that no conflict ever arises. In *conflict resolution*, on the other hand, no effort is made to avoid conflicts *ab initio*; instead, conflicts are resolved as and when they arise. The two methods may be thought of as eager and lazy implementations of conflict-handling. The next section considers these strategies in more detail, and introduces a hybrid strategy, employing elements of both, which is superior to either.

4. Algorithms for Register Allocation

This section describes three algorithms for register allocation and compares the quality of the code produced by each.

4.1. Algorithm I: Conflict Avoidance

As noted in the previous section, conflicts can arise during register allocation because a temporary variable may be allocated a register that is then needed as a parameter register before the last occurrence of that variable has been encountered. In the Conflict Avoidance strategy, all possible sources of conflicts are computed beforehand, and registers allocated so as to avoid conflicts. With each temporary variable is associated a set, its *CONFLICT* set, which contains those registers which could lead to a conflict if allocated to it. Clearly, conflicts can only arise from parameters in the last goal of a chunk. Therefore, temporary variables not occurring in the last goal of the chunk cannot give rise to conflicts. Finally, it is obvious that a variable can never be in conflict with itself. Thus, we have the following definition:

Definition: The set *CONFLICT*(T) associated with a temporary variable T in a chunk is defined as follows: if T occurs in the last goal of the chunk, then it is the set of argument positions in the last goal in which T does not occur at the top-level; otherwise, it is the empty set.

Example: Consider the clause

p(X,f(X),Y,W) :– X = [a | _], W > Y, q(Z,Y,X).

Then, *CONFLICT*(X) = {1,2}, *CONFLICT*(Y) = {1,3}, *CONFLICT*(W) = ∅.

The allocation process has two phases. In the first phase, the *USE*, *NOUSE* and *CONFLICT* sets are computed for each temporary variable by analysing variable occurrences in the source program. The second phase is part of the code generation phase, where the actual allocation of registers to temporary variables is done. Whenever a temporary variable T is encountered which has not had a register allocated to it, a register is allocated as follows: if a free register R that is in *USE*(T) can be found, it is allocated to T. Otherwise, a free register which is not in *NOUSE*(T) ∪ *CONFLICT*(T) is allocated.

This allocation procedure can be shown to be sound according to the criteria described earlier. We omit the proof of soundness here.

The principal advantage of Conflict Avoidance is its simplicity. However, as a result of the eager conflict-handling this strategy implements, it sometimes produces code that resolves conflicts too early, thereby leading to wasted effort. This is illustrated by the following:

Example: Consider the clause

p(X,Y,Z,a) :– q(Z,X,Y).

The code produced using conflict avoidance is

```
movreg 1,5
movreg 2,6
movreg 3,1
get_constant a,4
movreg 5,2
movreg 6,3
execute q/3
```

If, however, the instruction *get_constant a,4* fails, the three *movreg* instructions before it will have been executed to no avail. We turn our attention next to strategies that rectify this problem.

4.2. Algorithm II: Conflict Resolution

A second approach to handling conflicts is to take no steps *a priori* to prevent their occurrence, but to resolve them as and when they do happen. With this approach, therefore, *CONFLICT* sets are not computed. The algorithm involves two phases. During the first, *USE* and *NOUSE* sets are computed for each temporary variable; the second phase involves actual register allocation, conflict resolution and code generation. A table, *contents*, of register contents is maintained, indexed by register number, so that for any register we can determine its contents. Another table, *register*, is maintained, indexed on temporary variable name, and allows us to determine, given any temporary variable, which register is currently assigned to it.

The computation of *USE* and *NOUSE* sets in the first phase is straightforward. In the second phase of the algorithm, when a temporary variable is encountered in the process of code generation that has not been assigned a register, it is assigned a register as before: if a free register can be found that is also in the *USE* set of the variable, then this register is assigned to the temporary variable, otherwise a free register is assigned that is not in its *NOUSE* set. The tables *contents* and *register* are updated to reflect this. Code generation for all goals of the chunk except the last proceeds as for Conflict Avoidance. For the last goal of the chunk, however, when code is being generated for the $k^{th.}$ parameter par_k, if *contents[k]* is a temporary variable T and $T \neq par_k$, then a register R different from k, which is not in $NOUSE(T) \cup INUSE$, is chosen and the variable T is moved to register R (by generating a *‘‘movreg k, R’’* instruction), thereby resolving the conflict. The tables *contents* and *register* are updated accordingly: *contents[R]* is set to T, and *register[T]* is set to R. After this, code is generated for building and loading the $k^{th.}$ parameter into register k. The algorithm can be shown to be sound according to the criteria mentioned earlier.

Conflict Resolution represents a lazy approach to conflict handling, and thus avoids the problem of extra code executed because conflicts were resolved too early. Returning to the earlier example

p(X,Y,Z,a) :– q(Z,X,Y).

Here, *USE*(X) = {1,2}, *USE*(Y) = {2,3}, *USE*(Z) = {1,3}, *NOUSE*(X) = {3}, *NOUSE*(Y) = {1}, *NOUSE*(Z) = {2}. The inital register assignments are, therefore: X = 1, Y = 2, Z = 3 (recall that register 2 is not free when X is being assigned a register, so it is assigned the other regiter in its *USE* set, viz. register 2, and similarly for the other variables). Each temporary variable is left in the register it occurred in, and conflicts have to be resolved only after unification of the fourth parameter in the head succeeds. The code generated is

```
get_constant a,4
movreg 1,4
movreg 3,1
movreg 2,3
movreg 4,2
execute q/3
```

The code produced is clearly superior to that obtained from Conflict Avoidance.

The problem with the ‘‘pure’’ conflict resolution strategy described above is that there is no lookahead when a conflict is resolved, i.e. no attempt is made to see whether this could lead to further conflicts. This can result in cascading conflicts, whose resolutions incur a space and time penalty that might have been avoided. The following example ilustrates this:

Example: Consider the clause

p(X,a,b) :– q(c,d,f(X)).

Here, *USE*(X) = {1}, *NOUSE*(X) = ∅, and X is allocated register 1. When code for the call to q has to be generated, there is a conflict in the first parameter, so X is moved to the next available register, which happens to be register 2; at the next step, another conflict occurs, and X has to be moved again; and so on.

In the next section, we consider a third register allocation strategy, a hybrid of Conflict Avoidance and Conflict Resolution, which rectifies this problem. This turns out to be better than either of the two strategies considered so far.

4.3. Algorithm III: Mixed Conflict Resolution and Avoidance

When a conflict occurs between a temporary variable and a parameter, a new register has to be allocated in order to resolve the conflict. Thus, we are faced with a register allocation problem again. With pure conflict resolution, the Conflict Resolution strategy is applied again at this point. However, this may lead to recurring conflicts and result in inefficient code.

As noted earlier, the reason Conflict Resolution produces better code than Conflict Avoidance is that in Conflict Resolution, temporary variables tend to be left in registers as long as possible. What this means is that failures tend to occur earlier, so that fewer instruction executions are wasted. However, once a conflict occurs and the variable has to be moved to another register, it should be moved to a register where it will not generate any more conflicts. In this way, at most one *movreg* is required per temporary variable to handle conflicts, as in the case of Conflict Avoidance, but the execution of this *movreg* instruction is delayed as far as possible. In other words, when allocating a register to resolve a conflict, we should use the Conflict Avoidance strategy rather than apply the Conflict Resolution strategy recursively. Note also that the advantages of Conflict Resolution apply only to variables whose first occurrence is at the top level in the head of the clause, since in this case the variable is already in a register and need not be moved until a conflict occurs. If the first occurrence is not at the top level in the head, the variable has to be moved to a register anyway, and by the same argument as before, Conflict Avoidance is preferable in this case.

This suggests a hybrid allocation strategy. The data structures maintained are: for each temporary variable, its *USE*, *NOUSE* and *CONFLICT* sets; and two tables, *contents* and *register*, as for Conflict Resolution. The algorithm proceeds in two phases, as before. In the first phase, *USE*, *NOUSE* and *CONFLICT* sets are computed for each temporary variable. The second phase invloves the actual register allocation, code generation and conflict handling. When a temporary variable is encountered during code generation in this phase that has not been allocated a register, it is allocated a register as follows: if its first occurrence is at the top level in the head of the clause, then the register is allocated using Conflict Resolution, else the register is allocated using Conflict Avoidance. The tables *contents* and *register* are updated accordingly. Code generation for all goals in the chunk except the last proceeds as for Conflict Avoidance. For the last goal of the chunk, however, when code is being generated for the $k^{th.}$ parameter par_k, if *contents[k]* is a temporary variable T and $T \neq par_k$, then a register R different from k and which has been allocated using Conflict Avoidance is chosen, and the variable T is moved to it (by generating a ‘‘*movreg k, R*’’ instruction), thereby resolving the conflict. The tables *contents* and *register* are updated accordingly: *contents[R]* is set to T, and *register[T]* is set to R. After this, code is generated for building and loading the $k^{th.}$ parameter into register k. The algorithm is sound according to the criteria mentioned earlier.

The mixed strategy overcomes the deficiencies of the pure Conflict Avoidance and Conflict Resolution strategies, and produces code superior to that produced by either strategy. This is illustrated by the following example:

Example: Consider the clause

```
p(X,Y,f(Z)) :– q(a,b,Z,g(X,Y)).
```

The variables X and Y occur at the top level in the head, and are allocated according to the conflict resolution strategy. Thus, they are not moved until the unification of the third parameter in the head has succeeded. Then, conflicts are found to occur, and these are resolved using Conflict avoidance, so that further conflicts do not occur. The code generated is

```
get_structure f, 3
unify_temp_variable 3              ; USE(Z) = {3}
movreg 1, 5
put_constant a, 1
movreg 2, 6
put_constant b, 2
put_structure g, 4
unify_temp_value 5    ; X is in register 5
unify_temp_value 6    ; Y is in register 6
execute q/4
```

5. Global Register Allocation

The register allocation algorithms presented in the previous section were based on a definition of temporary variables that assumed that no information was available regarding the usage of registers by other predicates. This assumption constrains temporary variables to occur in at most one chunk in the body of a clause. Since such algorithms allocate registers using only local register usage information, we refer to them as *local allocation algorithms*. It is possible, however, to use register usage information across predicates to improve register allocation. We do not consider details of such *global allocation algorithms* here, but simply outline the basic idea behind such schemes.

If we consider the optimized code for *append/3* illustrated earlier, we notice that only registers 1 through 4 are used in this predicate. After any call to *append/3*, therefore, all registers other than 1 through 4 are unchanged. Once this information is available to the compiler, it can modify its register allocation in predicates that call *append/3* to take advantage of this fact. This is illustrated by the following example:

Example: Consider the clause

p(X,Y,Z) :– append(X,Y,W), q(W,Z).

In this case, the variable Z appears in two chunks, and is therefore a permanent variable. This means that, by our earlier definitions, it has to be kept in the activation record in memory rather than in a register. However, given the knowledge that *append/3* affects only registers 1 through 4, we can store Z in register 5, say, across the call to *append*, and restore it before the call to *q*. The code generated is:

```
movreg 3, 5             ; save Z in a safe register
put_perm_variable 2, 3
call append/3
put_perm_value 2, 1
movreg 5, 2             ; restore Z
execute q/2
```

This replaces register-memory moves by register-register moves, which are usually much cheaper, and also effects a reduction in the activation record size. In the general case, a permanent variable would be a candidate for this kind of global allocation if its first occurrence was in the head or in a structure. Since a decision to keep a permanent variable in a register affects the register usage of that predicate, the global allocation algorithm will have to take such changes into account when doing global register allocation in other predicates.

One simple scheme for global register allocation could proceed as follows: define a relation *is-called-by* between predicates as follows: p is called by q if p appears in the body of a clause for q, or there is some predicate r such that r is called by q and p is called by r. The program is partitioned into sets of predicate definitions $\{S_1, ..., S_n\}$ such that if $i < j$, then no predicate p defined in S_j is called by any predicate q defined in S_i. Global register allocation is performed on predicates in S_k only after it has been done for all predicates in $S_1 ... S_{k-1}$, with permanent variables being saved in registers across a call only if the call is to a predicate defined in S_j, for some $j < k$.

6. Summary

This paper considers the problem of efficient register allocation for temporary variables in the Warren Prolog Engine. What makes the register allocation problem different in this case is the parameter-passing convention used in the machine. The approach we take is high-level in the sense that most of the relevant information is derived from an analysis of the Prolog source program. Three register allocation algorithms are presented, representing, respectively, eager, lazy, and a hybrid approach to conflict handling (the allocation strategy used in the compiler developed at SUNY Stony Brook is a variant of the eager conflict handling algorithm). They are relatively simple to implement, and produce code that is quite good, and often optimal.

Acknowledgements

Thanks are due to David S. Warren for many helpful discussions, and to Jiyang Xu and David Znidarsic for their help and patience during debugging.

References

1. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers - Principles, Techniques and Tools*, Addison-Wesley, 1986.
2. V. Rajlich and M. D. Moshier, A Survey of Algorithms for Register Allocation in Straight Line Programs, Tech. Report CRL-Tech. Rep.-14-84, Univ. of Michigan Computing Research Lab., Ann Arbor, Mich., Feb. 1984.
3. P. Van Roy, A Prolog Compiler for the PLM, Report No. UCB/