Last Parallel Call Optimization and Fast Backtracking in And-parallel Logic Programming Systems

Tang DongXing, Enrico Pontelli
Gopal Gupta

Laboratory for Logic and Databases
Dept of Computer Science
New Mexico State University
Las Cruces NM USA
{dtang,epontell,gupta}@cs.nmsu.edu

Manuel Carro

Facultad de Informática
Universidad Politècnica de Madrid
28660-Boadilla del Monte
Madrid, Spain
mcarro@fi.upm.es

Abstract

In this paper we present a novel optimization called *Last Parallel Call Optimization.* The last parallel call optimization can be regarded as an extension of last call optimization, found in sequential systems, to and-parallel systems. The last parallel call optimization leads to improved time and space performance for a majority of and-parallel programs. The last parallel call optimization is presented in detail in this paper and its advantages discussed at length. The last parallel call optimization can be incorporated in a parallel system (such as RAPWAM) through relatively minor modifications to the runtime machinery. We also present some experimental results from a limited implementation of last parallel call operation done on the DDAS System. These experimental results prove that last parallel call optimization is indeed effective and produces better speed-ups with respect to an unoptimized implementation. We also discuss the problem of efficiently performing the *kill* operation in and-parallel systems. We present two approaches for efficiently propagating the kill signal to other parallel calls subsumed by the subgoal that received the kill signal. The first approach, implemented in the and-parallel component of the ACE system, propagates the kill *lazily* while the second one propagates the kill signal *eagerly.* The advantages and disadvantages of both these approaches are presented. The implementation and optimization techniques presented in this paper are very pragmatic and we believe that they will be of considerable utility to implementors of and-parallel systems.

1 Introduction

A distinguishing feature of logic programming languages is that they allow considerable freedom in the way programs are executed. This latitude permits one to exploit parallelism implicitly (without the need for programmer intervention) during program execution. Indeed, two main types of control parallelism have been identified and successfully exploited in logic programs:

(i). *Or-parallelism:* arises when more than a single rule define some relation and a procedure call unifies with more than one rule head—the corresponding bodies can then be executed in or-parallel fashion. Or-parallelism is thus a way of efficiently searching for solution(s) to the top-level query.

(ii). *And-parallelism:* arises when a set of conjunctive goals in the current resolvent are executed in parallel. The conjunctive goals could either be *independent*, i.e., the arguments of the conjunctive goals are bound to ground terms or have non-intersecting set of unbound variables (independent and-parallelism), or they could be *dependent* in which case they will be executed in parallel until they access the common variable (dependent and-parallelism).

In this paper we will focus on independent and-parallelism although our results are also applicable to dependent and-parallel systems and and-or parallel systems. A major problem in implementation of and-parallel systems is that of efficient implementation of backtracking. Because of and-parallelism not only a new backtracking semantics is needed for such systems, but also its implementation becomes very tricky. We consider the backtracking semantics given by Hermenegildo and Nasr for and-parallel systems [5] and its efficient implementation in RAPWAM [4]. The backtracking semantics as given by Hermenegildo and Nasr attempts to emulate the backward execution control of Prolog as much as possible. In this paper, we present some optimizations over this backtracking scheme that permit faster backward execution. In particular, we propose the Last Parallel Call Optimization that saves both time and space when the last call in a clause is itself a parallel conjunction (from now on a parallel conjunction will also be referred to as a *parcall* for brevity). Last Parallel Call Optimization (LPCO), when applicable, simplifies backtracking and allows failures and *kills* to be propagated faster. We present some experimental results to demonstrate the advantages of Parallel Last Call Optimization. We also discuss how the *kill* operation can be efficiently implemented in an and-parallel system. The kill operation needs to be performed to terminate computation in subgoals that constitute a parallel conjunction when one of the subgoals fails to produce a solution. Its implementation is complicated by the fact that a parallel conjunction may have other nested conjunctions inside that need to be recursively traversed and the computation in their and-parallel subgoals terminated. Because the kill operation involves a number of processors that simultaneously prune and modify a shared tree, a naive implementation may result in race conditions and non-terminating situations, as well as in inefficiency. In this paper we present two approaches for implementing a kill in an and-parallel system. One of the approach is *lazy* while the other is *eager* depending on the way in which killing information is propagated along the branches of the computation tree.

The rest of the paper is organized as follows: Section 2 describes the backtracking scheme of Hermenegildo and Nasr for independent and-parallel system and its realization in Hermenegildo's RAPWAM [4]. Section 3 introduces Last Parallel Call Optimization. Section 4 briefly describes an implementation scheme for the LPCO, while Section 5 describes our experiments to test the applicability of LPCO to existing systems. Section 6 describes in detail the *kill* operation, the problems in implementing it, and the various solutions that we have proposed. The implementation of kill as reported in this paper has been incorporated in the and-parallel component of ACE [8], an and-or parallel system. Section 7 presents our conclusions. We assume that the user has some familiarity with and-parallelism and and-parallel systems such as &-Prolog. We will illustrate our ideas and concepts in the context of independent and-parallelism with goal recomputation (goal recomputation means that subgoals to the right of another subgoal g in a parallel conjunction are computed in their entirety for every solution found for subgoal g), although our results are also applicable to dependent and-parallel systems such as DDAS [10] and to and-or parallel systems such as ACE [3].

2 Backtracking in And-parallel Systems

An and-parallel system works by executing a program that has been annotated with parallel conjunctions. These parallel conjunction annotations are either inserted by a parallelizing compiler [6, 7] or by the programmer. Execution of all goals in a parallel conjunction is started in parallel when control reaches that parallel conjunction. Parallel conjunctions may also be conditional, which means that the goals in the conjunction are executed in parallel only if the condition, i.e., the expression upon which the conjuction is conditioned, evaluates to true (e.g., Conditional Graph Expressions [2, 5]).

Backtracking becomes complicated in and-parallel system because more than one goal may be executing in parallel, one or more of which may encounter failure and backtrack at the same time. Unlike a sequential system, there is no unique backtracking point. In an and-parallel system we must ensure that the backtracking semantics is such that all solutions are reported. One such backtracking semantics has been proposed by Hermenegildo and Nasr: consider the subgoals shown below, where ',' is used between sequential subgoals (because of data-dependencies) and '&' for parallel subgoals (no data-dependencies):

```
a, b, (c & d & e), g, h
```

Assuming that all subgoals can unify with more than one rule, there are several possible cases depending upon which subgoal fails: If subgoal `a` or `b` fails, sequential backtracking occurs, as usual. Since `c`, `d`, and `e` are mutually independent, if either one of them fails, backtracking must proceed to `b`—but see further below. If `g` fails, backtracking must proceed to the right-most choice point within the parallel subgoals `c & d & e`, and recompute all goals to the right of this choice point. If `e` were the rightmost choice point and `e` should subsequently fail, backtracking would proceed to `d`, and, if necessary, to `c`. Thus, backtracking within a set of and-parallel subgoals occurs only if initiated by a failure from outside these goals, i.e., "from the right" (also known as *outside backtracking*). If initiated from within, backtracking proceeds outside all these goals, i.e., "to the left" (also known as *inside backtracking*). This latter behavior is a form of "intelligent" backtracking. When backtracking is initiated from outside, once a choicepoint is found in a subgoal g, an untried alternative is picked from it and then all the subgoals to the right of g in the parallel conjunction are restarted.

Independent and-parallelism with the backtracking semantics described above has been implemented quite efficiently in RAPWAM [4]. RAPWAM is an extension to the sequential WAM for and-parallel execution of Prolog programs with and-parallel annotation (such as CGEs [5]). In order to execute all goals in a parallel conjunction in parallel, RAPWAM has a scheduling mechanism to assign parallel goals to available processors and some extra data structures to keep track of the current state of execution. The two main additional data structures are the *goal stack* and the *parcall frame*. Details of the structure of a Parcall frame are shown in Figure 1 (more details can be found in [4, 5]). In addition to parcall frames and goal stacks, an *input marker node* and an *end marker node* is used to mark the beginning and the end respectively of the segment in the stack corresponding to an and-parallel goal.
During execution of and-parallel Prolog programs, when a parallel conjunction is reached that is to be executed in parallel (recall that a conditional parallel conjunction may be executed sequentially if the conditional test fails), a *parcall frame* is created in the local stack[1]. The parcall frame contains: (i) a slot for each goal in the parallel conjunction where information regarding the state of execution of that goal will be recorded, (ii) necessary information about the state of the execution of the parallel conjunction. After the parcall frame is created for a parallel conjunction, all the goals in the parallel conjunction are pushed into the *goal stack*. Each entry in the *goal stack* contains all the information required to allow a remote execution of the corresponding subgoal. Each processors can pick up a goal for execution from the *goal stacks* of other processors as well as their own goal stack, once they become idle.
The execution of a parallel conjunction can be divided into two phases. The first phase, called *inside phase*, starts with the creation of the parcall frame and ends when the execution of the continuation of the parallel conjunction is first begun (i.e. each goal in the conjunction has found its

[1]or in the choice point stack, as in ACE and DDAS.

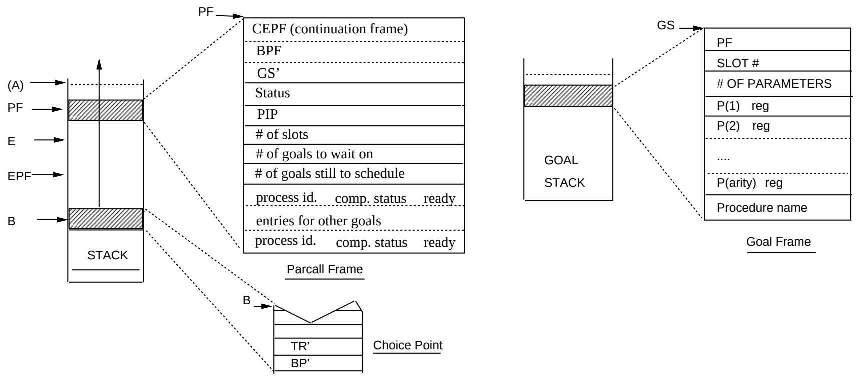


Figure 1: Additional data structures and related registers in RAPWAM

first solution; we will say that the parallel conjunction has been completed). Once the execution of continuation is begun for the first time, *outside phase* is entered. If failure occurs in the inside phase, inside backtracking is used, while if failure occurs in the outside phase outside backtracking will be used for backtracking on the parallel conjunction. In the *inside* phase, if a goal fails, the whole parcall should fail since all goals in the parcall are assumed independent. Therefore, when a goal fails in a parcall during the *inside* phase, the failing processor should send a kill signal to all processors that have stolen a goal from that parcall to undo any execution for the stolen goal. After all processors finish undoing the work, the goal before the CGE will be backtracked over as in standard WAM.

On the other hand, after a parallel conjunction completes, if a goal in the continuation of the CGE fails, then backtracking proceeds into the conjunction in outside mode. Outside backtracking is from right to left in the CGE similar to the backtracking in sequential WAM. The only difference is that a goal to be backtracked over may have been executed by a remote processor if another processor stole the goal. Thus the redo signal has to be sent to the remote processor. If a new solution is found during backtracking, the goals to the right of this goal in the parallel conjunction have to be reexecuted. If outside backtracking fails to produce any more answers, the goal before the CGE will be backtracked over as in normal sequential execution.

3 Last Parallel Call Optimization

Last Parallel Call Optimization produces the following advantages in an and-parallel system:

(i). It speeds up forward execution by avoiding allocation of certain parcall frames;

(ii). It speeds up the process of killing computations during an inside backtracking;

(iii). It speeds up the process of backtracking, in general;

(iv). It saves space on the stacks and allows earlier recovering of space on backtracking.

The advantages of LPCO are very similar to those for last call optimization [11] in the WAM. The conditions under which the LPCO applies are also very similar to those under which last call optimization is applicable in sequential systems.

Consider first an example that covers a special case of LPCO: `?- (p & q).` where

```
p :- (r & s).
q :- (t & u).
```

The and-tree constructed is shown in Figure 2(i). One can reduce the number of parcall nodes, at least for this example, by rewriting this example as `?- (r & s & t & u).` Figure 2(ii) shows the and-tree that will be created if we apply this optimization. Note that executing the and-tree shown in Figure 2.(ii) on RAPWAM will require less space because the parcall frames for `(r & s)` and `(t & u)` will not be allocated. The single parcall frame allocated will have two extra goal slots compared to the parcall frame allocated for `(p & q)` in Figure 2(i). It is possible to detect cases such as above at compile time. However, our aim is to accomplish this saving in time and space at runtime. Thus, for the example above, our scheme will work as follows. When the parallel calls `(r & s)` and `(t & u)` are made, the runtime system will recognize that the parallel call `(p & q)` is immediately above and instead of allocating a new parcall frame some extra information will be added to the parcall frame of `(p & q)` and allocation of a new parcall frame avoided. The extra information added will consist of adding slots for the goals `r`, `s`, etc. Note that no new control information need be recorded in the parcall frame of `(p & q)` (however, some control information, such as the number of slots, etc., need to be modified in the parcall frame of `(p & q)`).

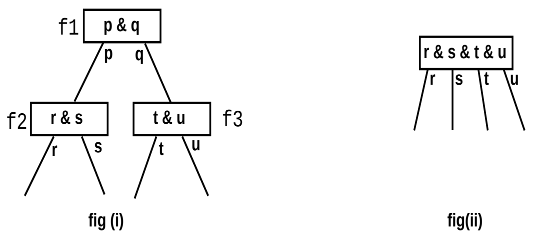


Figure 2: Reusing Parcall Frames

Note also that if the goal `r` was to fail in inside mode, then in case (ii) (Figure 2(ii)) killing of computation in sibling and-branches will be considerably simplified. In case (i) the failure will have to be propagated from parcall frame f2 to parcall frame f1. From f1 a kill message will have to be sent out to parcall frame f3.

One could argue that the improved scheme described above can be accomplished simply through compile time transformations. However, in case `p` and `q` were dynamic predicates this would not be possible. Also, for many programs the number of parallel conjunctions that can be combined into one will only be determined at runtime. For example, consider the following program:

```
process_list([H|T], [Hout | Tout]) :- (process(H, Hout
process_list(T, Tout)).  process_list([], []).

?-process_list([1,2,3,4], Out).
```

In such a case, compile time transformations cannot unfold the program to eliminate nesting of parcall frames because it will depend on the length of the input list. However, using our runtime technique, given that the goal `process` is determinate, nesting of parcall frames can be completely eliminated (Figure 3). As a result of the absence of nesting of parcall frames, if the `process` goal fails for some element of the list, then the whole conjunction will fail in one single step.

Efforts have been made by other researchers to make execution of recursive program such as above more efficient. Heremenegildo and others have suggested partially unfolding the program so that instead of allocating one parcall frame per recursive call, one is allocated per n calls, where n is the degree of unfolding as illustrated in the code below ($n = 3$).

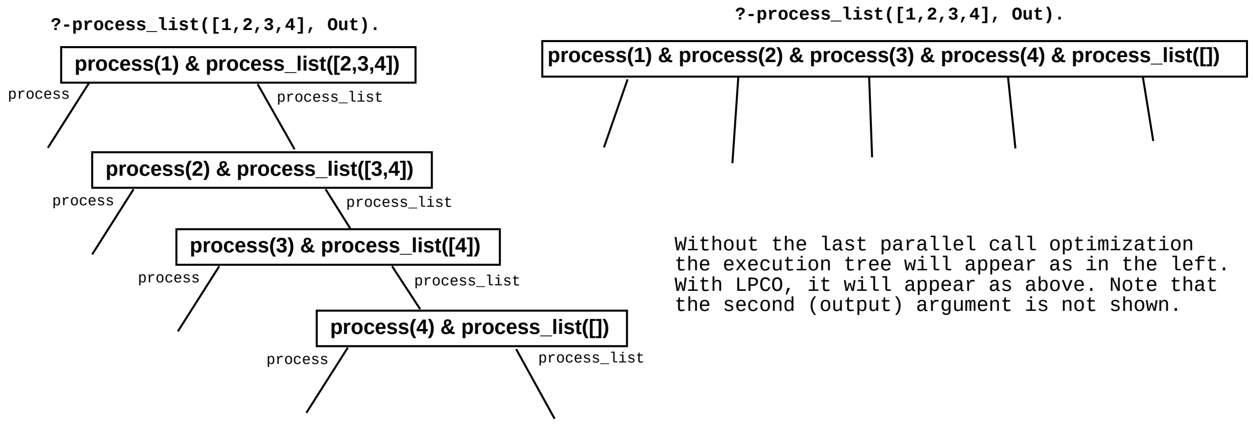
```
process_list([X,Y,Z|T],[Xo,Yo,Zo|Tout]):- (process(X,X
process(Y,Yo) & process(Z,Zo) & process_list(T,Tout))

process_list([X,Y], [Xo,Yo]):- (process(X, Xo) & proce

process_list([X], [Xo]):- process(X, Xo).

process_list([], []).
```

Barklund et al have suggested new language constructs (the language augmented with these constructs is termed Reform Prolog [1]) based on *Bounded Quantification* that encapsulate a call such as `process_list(Lin, Lout)` in such a way that it is executed in parallel in one (parallel) step.



?-process_list([1,2,3,4], Out).
process(1) & process_list([2,3,4])
process
process_list
process(2) & process_list([3,4])
process(3) & process_list([4])
process(4) & process_list([])
?-process_list([1,2,3,4], Out).
process(1) & process(2) & process(3) & process(4) & process_list([])

```
Without the last parallel call optimization
the execution tree will appear as in the left.
With LPCO, it will appear as above. Note that
the second (output) argument is not shown.
```

Figure 3: Reuse of Parcall Frames for Recursive Programs

Compared to Bounded Quantification and program unfolding, our technique based on last parallel call optimization does not require any programmer intervention or pre-processing by a compiler and achieves optimal saving in space and time.

Next we present the most general case of LPCO. The most general case of LPCO arises when there are goals preceding the parallel conjunction in a clause that matches a subgoal that is itself in a parallel conjunction. Thus, given a CGE of the form: `(p & q)` where

```
p :- e, f, g, (r & s).  q :- i, j, k, (t & u).
```

LPCO will apply to `p` (resp. `q`) if

- There is only one matching clause for `p` (resp. `q`), i.e., `p` (resp. `q`) is determinate.
- All goals preceding the parallel conjunction in the clause for `p` (resp. `q`) are determinate.

If these conditions are satisfied then a new parcall frame is not needed for the parallel conjunction in the clause. Rather the parcall frame for `(p & q)` can be extended with an appropriate number of slots and execution continues as if clause for `p` was defined as `p :- ((e,f,g,r) & s).` Thus, if we determine at the time of the parallel call `(r & s)` that `e`, `f`, and `g` are determinate then we pretend as if the clause for `p` is defined as `p :- ((e,f,g,r) & s).` This is illustrated in Figure 4.

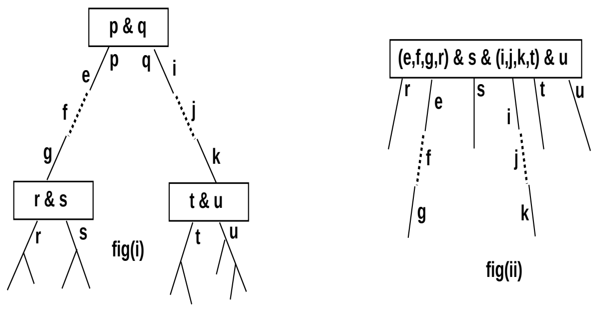


Figure 4: Last Parallel Call Optimization

Note that the two determinacy conditions above require that when the parallel conjunction is encountered at the end of clause for `p` then there are no intervening choicepoints between the parcall frame for `(p & q)` and the current point on the stack. Thus even though goal `p` is not determinate in the beginning, the determinacy conditions will be satisfied when the last clause for `p` (resp. `q`) is tried. LPCO can be applied at that point. This is akin to last call optimization in sequential systems when even though a goal is not determinate, last call optimization is triggered when the last clause for that goal is tried. Note also that the conditions for LPCO do not place any restrictions on the nature of goals in the clause for `p` (resp. `q`). The goals `r`, `s`, etc. can be non-deterministic. Note that when outside backtracking takes place in the tree in Figure 4(ii), then because of the organization of the parcall frame, backtracking will directly proceed into goal `s` from goals `t` and `u`. Backtracking over goals `i, j, k` will be missed. Suppose now an untried alternative is found within `s`, then the execution of goals `t` and `u` has to be restarted. At this point because goals `i, j, k` were never backtracked over, their existing computation can be reused, thanks to their determinacy. However, when we completely backtrack out of the parcall frame, then care has to be taken that trail sections corresponding to to `i, j` and `k` (as well as `e, f,` and `g`) are unwound.

Finally, note that LPCO can be generalized further. Given a parallel conjunct `(p & q)` and the clause `p :- e, f, g, (r & s), h`, then it is still possible to avoid allocation of the parcall frame for `(r & s)`, augmenting the parcall frame of `(p & q)` instead, if goals in the continuation of `(r & s)`, i.e., `h` in this example, are determinate. However, determinacy of the continuation of the parallel conjunct will have to be known in advance, hence some kind of static analysis will have to be used to collect this information. In this paper we do not consider this optimization any further.

4 Implementation of LPCO

To implement LPCO, the compiler will generate a different instruction when it sees a parallel conjunct at the end of a clause. This instruction behaves the same as `alloc_parcall` instruction of the RAPWAM, except that if the conditions for LPCO are fulfilled last parallel call optimization will be applied.

Thus, first the code for this instruction will check if there are any choicepoints below the immediate ancestor parcall frame (pointed to by PF register of RAPWAM). If there are no choicepoints, then the determinacy condition is satisfied and LPCO can be applied.

To apply LPCO, the immediate ancestor parcall frame (or immediately enclosing parcall frame) will be accessed and if the current parallel conjunction has n and-parallel goals, then n new slots corresponding to these n goals will be added to it. The number of slots should be incremented by n in the enclosing parcall frame (this operation should be done atomically).

Recall that in traditional RAPWAM the slots for goals are part of the parcall frame that resides on the stack. Given that the enclosing parcall may be allocated somewhere below in the stack, adding more slots to it may not be feasible. To enable more slots to be added later, the slots will have to be allocated on the heap and pointer to the beginning of the slot list stored in the parcall frame (Figure 5). The slot list can be maintained as a double linked list. Also, each input marker of an and-parallel goal has a pointer to its slot in the slot list for quick access. With the linked list organization, adding new slots becomes quite simple as shown in Figure 5. Note that modification of the slot list will have to be an atomic operation. The enclosing parcall frame becomes the parcall frame for the last parallel call, and rest of the execution will be similar to that in standard RAPWAM.

Note that changing the representation of slots from an array recorded on the stack (inside a parcall frame) to a linked list on the heap will not add any inefficiency because an and-parallel goal can access its corresponding slot in constant time via its input marker.

It is obvious that LPCO indeed leads to saving in space as well as time during parallel execution. Space (as well as time) is saved because allocation of parcall frames can be avoided. Time is also saved because backtracking and kill become faster: there are fewer parallel control structures (parcall frames) on the stack simplifying backward and forward control.

5 Experimental Results

We implemented our ideas described above on the emulator of the DDAS [10] system. In fact because the implementation was unfamiliar to us, we implemented a diluted form of LPCO (described below). Even with this restricted imple-

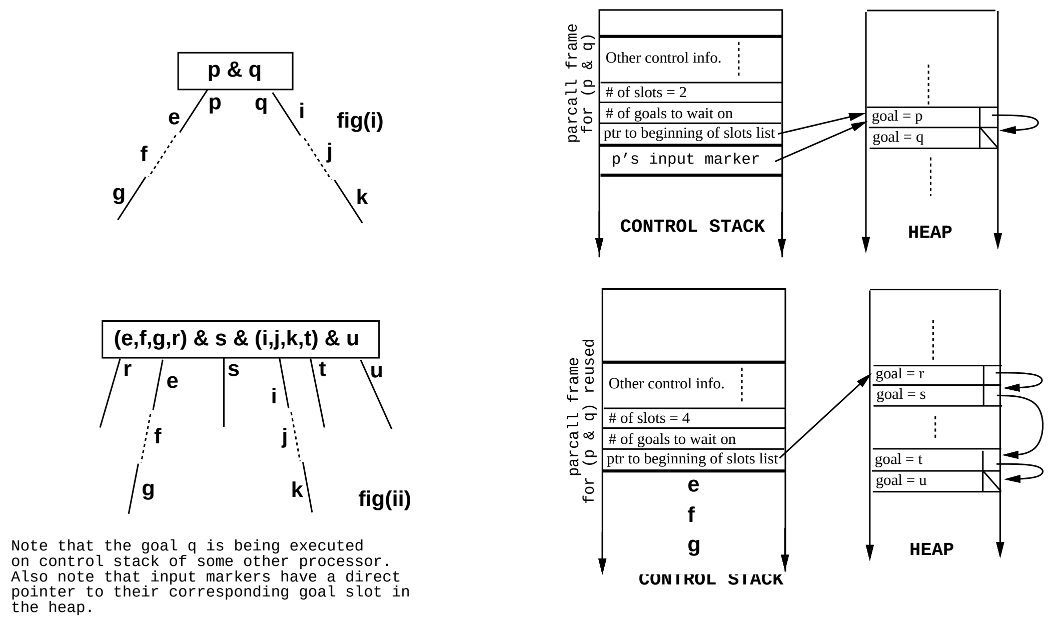

p & q
fig(i)
(e,f,g,r) & s & (i,j,k,t) & u
fig(ii)
Note that the goal q is being executed on control stack of some other processor. Also note that input markers have a direct pointer to their corresponding goal slot in the heap.
parcall frame for (p & q)
Other control info.
# of slots = 2
# of goals to wait on
ptr to beginning of slots list
p's input marker
CONTROL STACK
HEAP
goal = p
goal = q
parcall frame for (p & q) reused
# of slots = 4
goal = r
goal = s
goal = t
goal = u

Figure 5: Allocating Goal Slots on the Heap

mentation of LPCO we obtained improved speed-ups for all examples that we tried. Essentially, the deeper the nesting of a CGE in the benchmark the more improvement we obtained. Deep levels of nesting are not uncommon in and-parallel programs (e.g., in matrix multiplication, levels of nesting of the order of thousand parcalls can be created).

Our diluted implementation of LPCO was as follows: When a goal in a CGE C is stolen from the goal stack of another processor, and the execution of this stolen goal deterministically reaches another CGE C', then we *chain* the parcall frame for the current CGE C' to the parcall frame corresponding to the CGE C (In a more faithful implementation of LPCO the parcall frame for C' would not have been allocated; the parcall frame for C would have been partially expanded). Having maintained such *parcall frame chain(s)*, when the execution of a CGE corresponding to some parcall frame in the chain fails, we send a kill signal to all the processors which are executing or have executed a goal taken from a parcall frame that is in this chain. In this way the kill signals reach the processor in the minimum amount of time possible. In the standard RAPWAM, the kill signals will be sent gradually as the failing processor backtracks over the search tree. As a result, not only processors do less useless work that will be eventually killed, they undo the work that needs to be undone and execute other goals that may lead to useful solutions sooner. This results in execution speed-ups.

Consider the following program.

```
?- g.                          a.
g :- (a & b & c & d).          d.
b :- (x & y).                  x.
c :- (u & v).                  u.
y :- (e & f).                  v.
```

Assume there are four processors (say P1, P2, P3, and P4), all parallel conjunctions can be executed in parallel, and the top-level query `?- g` is executed by P1. When P1 reaches the parallel conjunction `(a & b & c & d)`, a parcall frame will be created in its local stack S1 and then all goals in the CGE will be pushed into its goal stack G1. Assume that P1 executes the first goal `a` locally (indicated by * in Figure 6), P2 steals the goal `b` from G1 for (remote) execution, and P3 steals the goal `c` for (remote) execution. During the execu-

tion of P2, a parcall frame will be created in the local stack S2 when it reaches the CGE (x & y). Since the execution leading to this parcall frame is deterministic, pointers are created linking the parcall frame for (a & b & c & d) and the parcall frame currently created for (x & y). Similar situation occurs with respect to execution of goal c by processor P3 (Figure 6).
During the execution of (x & y), assume that P2 continues to execute the first goal x and P4 steals goal y for remote execution. Execution of goal y will determinately lead to CGE (e & f). Thus, a parcall frame is created for this CGE in stack S4 of P4 and chained to the parcall frame for the CGE (x & y) in S2. Finally, when P4 executes the goal e, execution fails. Therefore, a kill signal will be sent by P4 to processors P1 and P2 (note that P4 could also send a kill signal directly to processor P3 as well but because of lack of complete understanding of code for DDAS system we could not implement it in our modification; P3 will receive a kill signal from P1 via standard kill mechanism employed by DDAS), since they have executed goals taken from the *parcall frame chain*, to undo all work done corresponding to the chained parcall frames. After P1, P2, and P4 finish killing, they can find other work that may be useful.

In standard DDAS (or RAPWAM) implementation P4 will kill the computation corresponding to goals e and f. It will then backtrack to the parcall frame of (x & y) communicating the failure of that parcall frame to processor P2. P2 will then kill the computations for x and y and backtrack further up eventually communicating the failure of (a & b & c & d) to processor P1. As is apparent, the kill gets processed in a somewhat sequential manner in the RAPWAM, whereas due to LPCO it is propagated faster in the RAPWAM with our modification.

Our experimental results for three programs are shown in Figure 7 in which the dashed curves represent normal DDAS execution while the solid curves represents speed-ups on DDAS that includes our approximation of LPCO.

Figure 7(I) shows the speed-up in the execution of

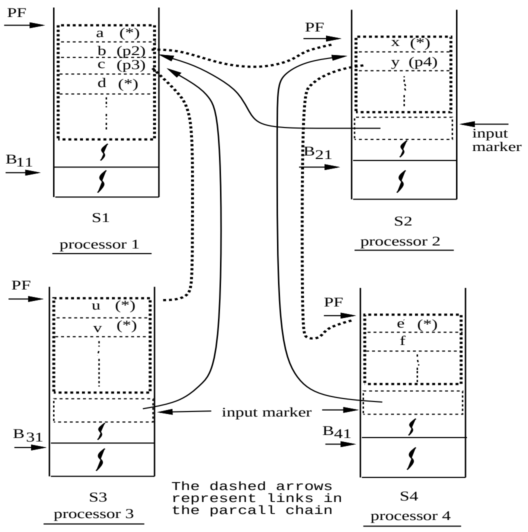


Figure 6: The *parcall frame chain*

.

a program that tests if an element is non-zero in a 250-element list, if yes, some processing is done, otherwise the computation fails (thus the program is similar to process_list program given in Section 3). Figure 7(II) is the speed-up of the execution of a program which checks if a given element is absent in a sorted binary tree. If the element is found, execution fails. Figure 7(III) is the speed-up graph of quicksort using difference list where both arguments of quicksort are ground. From the speed-up curves one can see that DDAS augmented with our approximation of LPCO performs consistently better than simple DDAS execution. In fact, we believe that if LPCO was implemented more faithfully, speed-up improvements will be even more dramatic (this is because in our diluted implementation we still incur the cost of allocating full parcall frames while in a more faithful implementation of LPCO this cost will be considerably reduced). The fact that LPCO leads to considerable improvement in execution speed-up is not all that surprising since Last Call Optimization also results in dramatic improvement in execution performance of sequential systems. Note that in Figure 7(III) the fact that the dashed curve and dotted curves meet is because all available parallelism has been ex-

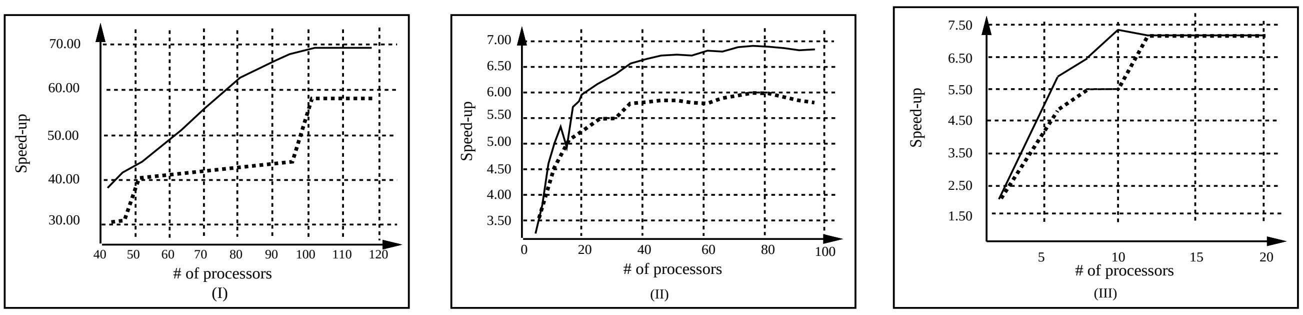


Figure 7: Experimental Results

ploited when that point is reached in which case LPCO cannot produce any performance improvement.

6 Implementing *KILL* in And-parallel Systems

In this section we discuss two methods for implementing the *kill* operation. As mentioned earlier, performing the kill is not easy since it is a global operation on the execution tree that may involve more than one processor.

The killing of a subgoal G involves complete removal of all the information allocated on stacks of processor(s) during the execution of G. It consists of two actions:

- **garbage collection**: recovery of the stack space occupied by the killed computation;
- **trail unwinding**: removal of all the bindings generated during the killed computation.

Both the actions do not impose any sort of constraint on the order in which they must be performed (i.e. the various parts of the computation may be deallocated and unwound in any order). However, care must be taken while mixing forward and backward execution. For instance, Consider a goal:

```
:- ....  a, (b & c & d) ...
```

in which the processor P1 that executed goal `a` also picks up goal `b`. Goals `c` and `d` are picked up by other processors. Suppose goal `b` fails, then processor P1 will send a kill to processors executing `c` and `d`. While the kill of `c` and `d` is in progress, P1 cannot backtrack over `a`, unwind the trail, and restart some other computation. In other words the processor executing the goal preceding the parallel conjunction should not restart computation elsewhere because unless `c` and `d` are completely killed the correct state would not have been restored to begin this new computation.

A kill phase is always started by a failing worker that reaches a parcall frame during backtracking. This covers two cases:

- the parcall frame is reached while there are other and-parallel subgoals of this parcall that are still active (i.e. the parcall frame is in *inside* status). In this case all the other subgoals of the parcall need to be killed.
- the parcall frame is reached after all the subgoals have detected at least one solution (i.e. the parcall frame is in *outside* status). In this case the traditional &-Prolog point-backtracking is applied and kill messages are sent to those and-parallel subgoals whose computation was deterministic (i.e., these determinate goals do not offer any further alternatives, and hence such goals should be killed immediately rather than backtracked over).

In the second case, all the processors involved in the kill operation are free to return to their normal previous operation once the kill is completed (since the worker generating the kills is itself taking care of continuing the execution by sending *redo* messages to the non-deterministic subgoals). In the first case, instead, once the kill is completed, one of the workers who was executing the parcall's subgoals needs to continue the main execution by continuing backtracking over the computation preceeding the parcall frame.

A kill can be serviced *lazily* or *eagerly*. Both approaches require different kind of support from the underlying runtime system. The only data structures that are common to both the lazy approach and the eager approach described in this paper are those that are required to support the sending/receiving of kill messages. Kill messages are realized by associating a kill-message queue with each worker. The kill-message queue of a processor is accessible to all other processors. Access operations on these queue must be atomic since these queues are shared.

6.1 Kill Steps

The process of killing a computation can be further subdivided into two distinct phases:

(i). **Propagation phase:** in which the kill signal is propagated to all the and-parallel branches nested inside the and-branch of the subgoal being killed;

(ii). **Cleaning phase:** in which the space from killed computation is removed (garbage collection and trail unwinding).

The execution of the cleaning phase is relatively easy but it requires the knowledge of the physical boundaries of the computation to be removed. The stack structure adopted to store the computation by any Prolog Inference Engine allows exclusively a bottom-up traversal of the computation tree (i.e. we can only visit the computation tree starting from the leaves and moving upwards, towards the root), which corresponds to scanning the stack from the top towards the bottom[2]. Thus to scan the tree, we *at least* require pointers to the bottommost leaf nodes that represent the point from which the upward traversal to clean up should begin. Once this starting point is known, cleaning is a straightforward operation, which resembles in many aspects a backtracking process. As in backtracking, the worker performing the kill scans the stack, removing each encountered data structure and unwinding the part of trail associated with that part of the computation. The main differences with the backtracking process are:

- alternatives in the choice points are ignored—and the choice points are removed;
- parcall frames are treated as if they are in *inside* status, i.e. kills towards all the subgoals of the parcall frame are generated.

It is important to observe that the cleaning activity can be performed quite efficiently since parallel branches enclosed in a killed subgoal can be cleaned in parallel. Once the bottommost extreme of the computation to be killed has been detected, the cleaning step can be immediately applied. Figure 8 shows this process. The main issue—and the most difficult problem—is the actual detection of the location of the leaves from which cleaning activity can be started. This is the purpose of the *propagation step* mentioned earlier and the rest of the section will deal with different approaches to tackling this problem.

In the following text we present two approaches for propagating kills (with possible variations). These approaches are parameterized by:

(i). **direction of the propagation:** two possible directions can be considered

   (a) *top-down*: kill signals are *actively* propagated from the root of the killed subgoal to the leaves;

   (b) *bottom-up*: kill is started from the leaves and pushed towards the root of the subgoal.

   Note that a *top-down* element is always present in any kill propagation mechanism since, after all, a kill is received by a subgoal

[2]a scan in the opposite direction would be very expensive, due to the variabile size of the structures allocated on the choice point stack.

and has to be propagated to its descendent parcall frames.

(ii). **mode of propagation:** the propagation of the kill signals in the tree can be realized in two alternative ways:

(a) *active*: the various workers are actively receiving and propagating the kill signals;

(b) *passive*: workers lazily wait to receive a kill directed to them.

6.2 Lazy Propagation of Kill Message

The main idea behind this propagation technique is to avoid sending kill messages (unless they are strictly necessary). This is realized by leaving to each processor the task of realizing when the computation that it is currently performing has been killed.

The only data structures that are required in order to implement lazy propagation of kill messages are the following:

(i). the generic support for the sending/receiving of kill messages such as kill-message queues, locks, shadow registers, etc.;

(ii). a unique global queue in which suspended kills are recorded;

(iii). a flag in each slot of the parcall frame which will be used to indicate that the corresponding subgoal should be killed;

(iv). a representation of the computation tree that will help determine efficiently whether a given subgoal is contained in another subgoal[3].

In the lazy approach to killing, a kill message is sent to a worker *only* when the bounds of the computation are known (i.e. the computation to be killed has already been completed). In this case the cleaning step can be immediately applied.

[3]It is an open question whether it is possible to obtain this information in constant time for a dynamically growing tree.

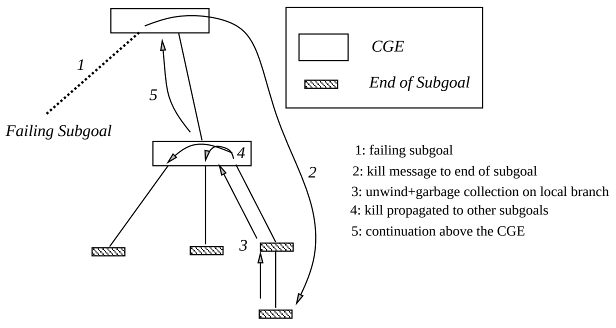

CGE
End of Subgoal
Failing Subgoal
1
2
3
4
5
1: failing subgoal
2: kill message to end of subgoal
3: unwind+garbage collection on local branch
4: kill propagated to other subgoals
5: continuation above the CGE

Figure 8: Cleaning Operation during a Kill

If the kill is issued when the branch to be killed is still computing, then a *suspended kill* is generated. The suspended kill is represented by

- setting the failing flag in the slot corresponding to the killed subgoal (i.e. subgoal corresponding to the killed branch);
- storing the information relative to the kill in a new entry of the global queue for suspended kills.

The effects of this operation are:

(i). since the worker which completes the execution of the subgoal will access the slot for updating the various field of the slot (like recording its id for backtracking purposes), it will immediately realize that the computation that it has just completed has been previously killed and automatically it will start performing the cleaning operation (as explained above).

(ii). if the and-scheduler selects an and-parallel subgoal that is subsumed by another subgoal with a suspended kill then it will immediately discard the goal and look for a new work. Key to this step is the presence of a representation of the computation tree which allows us to efficiently determine whether one subgoal is subsumed by another (i.e. one is a descendent of the other in the search tree).

(iii). periodically each worker checks whether its current computation is subsumed by one of the goals killed by a suspended subgoal. If this condition is satisfied then the worker will immediately interrupt the computation and start the cleaning phase.

The beauty of this approach lies in its simplicity. The scheme can also take advantage of many of the algorithms that have been developed for efficient backtracking (lazy kill is almost identical to backtracking). Furthermore, a worker is never distracted by kill messages during a useful computation, since the checks performed will affect its execution only if the worker is positioned on a killed branch of the tree. In this way the kill operation is postponed and performed only when no useful work is available.

The main disadvantages that we can identify in this approach are the following:

(i). the implementation of this scheme relies on the availability of a representation of the computation tree which allows to determine efficiently whether a given subgoal is a descendent of another. It is an open problem whether this can be done in constant time.

(ii). the execution of the kill may be slower than in other schemes; this is due to the fact that cleaning is started by one processor from the bottommost end of a branch, making it an inherently sequential operation. Other approaches may offer an higher degree of parallelism during the cleaning up of execution.

6.3 Eager Kill

The disadvantages mentioned above seems to make the Lazy Kill approach not too easily implementable. For this reason we propose a different approach, called eager kill, which is mainly (but not exclusively) a **top-down** approach (see Fig. 9).

The main problem in this approach is the lack of information that will allow us to perform a top-down traversal of the tree (starting from a given node towards the leaves). As we will see later on, the amount of information required to accomplish this for our purpose is quite limited.

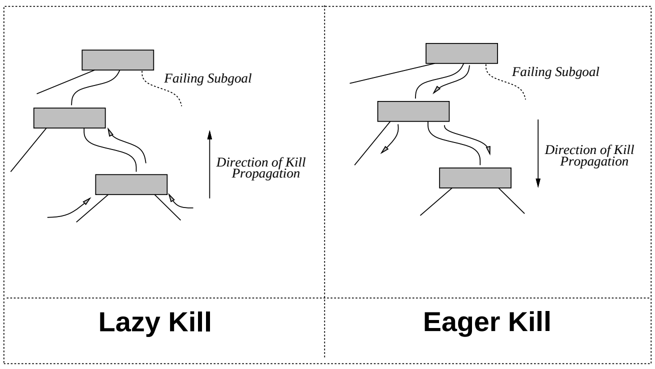


Figure 9: Comparison of Lazy and Eager Kill

6.3.1 Support for Eager Kill

In terms of data structures required to support eager kill, the following elements need to be added in the design:

- each slot is extended in order to contain a pointer either to (i) the first parcall frame (if any) created during the execution of the subgoal, (ii) or to the marker indicating the end of the computation, if this computation completes witout creating a parcall frame.
- a *return id* field is introduced in each parcall frame. This field will be used to indicate which worker is assumed to continue the kill/backtracking above the parcall frame in the execution tree once the whole computation originating from the parcall has been removed.

Note that the pointer to the first parcall frame created during the execution of a subgoal is indeed the same pointer needed to maintain the parcall chain described in Section 5

We now present an example to illustrate our technique for eager propagation of a kill.

6.3.2 An example

Let us consider the computation described in Figure 10.

Assuming that processor P_i is the one which started the execution of b, then the initial kill message will be sent to P_i from the worker which

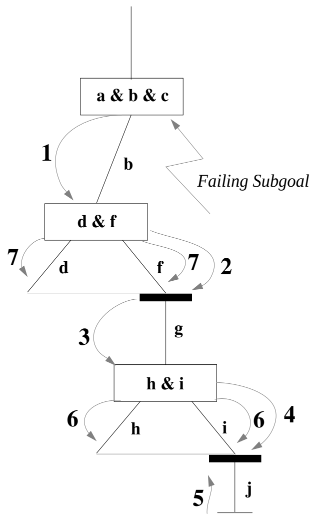


Figure 10: An Example of Eager Kill

failed in the computation of c. If P_i was looking for work by invoking the and-scheduler, then it will simply leave the and-scheduler and start serving the kill. Otherwise, at the next check for kill it will suspend the current execution, set the shadow registers and move to service the kill. Let's assume the second case.

As mentioned in section 6.3.1, P_i has access to the first parcall frame generated during the computation of b. It positions itself on that parcall frame and, since this has already completed (i.e. it is in outside status), it starts the killing activity by sending a kill to the continuation of the parcall frame (step **2**). The continuation itself contains another parcall frame (pointed to by the endmarker of the previous parcall frame), so the worker receving this kill message will access the parcall frame (step **3**) and send another kill message to its continuation (since even this parcall frame is in outside status). The worker executing j will receive the kill and serve it, removing the whole computation of j and setting the appropriate bit in the parcall frame ($h\&i$). At this point the worker that is busy waiting on such parcall frame (busy waiting until the continuation has been killed) will kill all the subgoals of the parallel call (h and i, step **6**) and then continue further and kill g. Once g has been removed, a bit in the parcall frame ($d\&f$) is set and P_i, which was in the meantime busy waiting on that parcall frame (busy waiting until the continuation has been killed), may proceed to send the kill messages to the subgoals of the parallel call (d and f, step **7**) and, once all of them have reported the end of the kill, it may proceed with the killing of b.

Once the whole branch has been removed and also a has reported the end of the kill, the worker P_i is free to restart the computation previously interrupted.

Note that both the Lazy and Eager schemes for propagating kill can be optimized further, however, we do not describe these possible improvements due to lack of space. More details can be found elsewhere [9]. The Lazy scheme has been incorporated in the and-parallel component of the ACE system [8, 3].

7 Conclusion

In this paper we presented a novel optimization called Last Parallel Call Optimization. The Last Parallel Call optimization can be regarded as an extension of last call optimization, found in sequential systems, to and-parallel systems. We also presented some experimental results that demonstrate the effectiveness of this optimization. Not only the LPCO saves space, it also leads to reduced runtime for a majority of and-parallel programs. The modifications needed to incorporate the LPCO in an and-parallel system are quite minor and only require some changes to the way the parcall frame (of RAPWAM) is implemented. We plan to include LPCO in the and-parallel component of ACE, an and-or parallel system being collaboratively developed by New Mexico State University and University of Madrid. We also discussed the problem of efficiently supporting the kill operation in an and-parallel system. We presented two approach, one *lazy* and the other *eager*, of which the former has been currently incorporated in the and-or parallel system ACE. The techniques discussed in this paper, we believe, are very pragmatic and will be immensely useful to implementors of and-parallel systems.

References

[1] J. Barklund, H. Millroth. Providing Iteration and Concurrency in Logic Program through Bounded Quantifications. In *Proc. International Conf. on Fifth Generation Computer Systems*, June 1992, pages 817–824.

[2] D. DeGroot. Restricted AND-parallelism. In *International Conference on Fifth Generation Computer Systems*, Nov., 1984.

[3] G.Gupta, E. Pontelli, M. Hermenegildo, V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proc. International Conference on Logic Programming*, 1994, MIT PRess, to appear.

[4] M. V. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Third International Conference on Logic Programming*, Lecture Notes in Computer Science 225, pages 25–40. Springer-Verlag, July 1986.

[5] M. V. Hermenegildo, R. I. Nasr, Efficient Implementation of backtracking in AND-parallelism. In *3rd International Conference on Logic Programming*, London, 1986. pages 40–54.

[6] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming.* MIT Press, October 1989.

[7] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):315–347, July 1992.

[8] E. Pontelli, G. Gupta, M. Hermenegildo. ACE: A progress Report. Technical Report. Department of Computer Science. New Mexico State University, March 1994.

[9] E. Pontelli, M. Carro, G. Gupta, "Kill and Backtracking in And-parallel Systems," Internal Report, ACE Project, Department of Computer Science, New Mexico State University, Dec. 1993.

[10] K. Shen: *Studies in And/Or Parallelism in Prolog.* Ph.D thesis, University of Cambridge, 1992.

[11] D. H. D. Warren. Last Call Optimization. "An Improved Prolog Implementation Which Optimises Tail Recursion," In *2nd International Logic Programming Conference*, 1984, K. Clark and S. A. Tärnlund (eds). Academic Press. Also Research Paper 156, DAI, Univ. of Edinburgh, 1980.