Determinacy Driven Optimization of And-parallel Prolog Implementations

Enrico Pontelli, Gopal Gupta, and Dongxing Tang
Laboratory for Logic, Databases, and Advanced Programming
New Mexico State University,
Las Cruces, NM, USA 88003
{epontell,gupta,dtang}@cs.nmsu.edu

Abstract

And-parallelism arises in Prolog programs when conjunctive subgoals in a query or the body of a clause are executed in parallel. In this paper we present three optimizations, namely, the *last parallel call optimization*, the *shallow parallelism optimization*, and the *processor determinacy optimization* that take advantage of determinacy to improve efficiency of and-parallel execution of Prolog programs. All three optimizations depend on *a posteriori* knowledge of determinacy rather than on *a priori* knowledge detected at compile-time. With the help of these optimizations, data-and parallel Prolog programs can be efficiently executed on very general and-parallel system (such as &ACE and &-Prolog) with the efficiency of dedicated data-and parallel systems (such as Reform Prolog system). These optimizations have been implemented in the &ACE system, and the results are also presented.

1 Introduction

Two main types of (implicit) and-parallelism have been identified and successfully exploited in logic programs:

(i). *Independent And-parallelism* arises when more than one goal is present in the query or in the body of a procedure, and the run-time bindings for the variables in these goals are such that two or more goals are *independent* of one another, i.e., their resulting argument terms after applying the bindings previously produced are either variable-free or have non-intersecting sets of variables.

(ii). *Dependent And-parallelism* arises when mutually dependent goals (i.e., goals that have variables in common) are executed in parallel to cooperatively produce a binding for these variables. Dependent and-parallelism is readily found, for example, in applications that involve producer-consumer interactions.

One of the facts well-known to implementors of logic programming systems is that determinacy of goals (i.e., at most one solution for a goal) can be used to considerably improve the performance of program execution. Thus, a number of optimizations that take advantage of determinacy for making sequential execution of Prolog faster (and, often, cheaper in terms of memory usage) have been proposed [2, 5]. Determinacy of goals has also been used in parallel systems: it is fundamental to the *Andorra Principle* [4], where it is used not only for reducing the search space of a program but also for exploiting parallelism, producing impressive results. The realization that determinacy plays a major role in system performance, has resulted in considerable amount of research in area of automatic compile time detection of determinacy for building "smart" compilers capable of generating efficient code [4, 10]. However, compile-time techniques have many limitations. Given that the compile-time inference of any non-trivial property of a program cannot be done with 100% precision, a compile-time approach will not be able to catch all the instances where an optimization is applicable. This justifies the need of having *run-time* optimizations.

In this paper we present three *novel* runtime optimizations for (general) and-parallel logic programming systems that are based on various notions of determinacy. These optimizations, are termed the *last parallel call optimization*, the *shallow parallelism optimization*, and the *processor determinacy optimization*, respectively. The first two of them are trigerred by the *determinacy of goals* (i.e., goals having at most one solution) while the third one takes advantage of the *determinacy of processors* (i.e., knowledge of the processor which will execute the goal). All three optimizations are fairly general in nature and are applicable to any parallel system that provides for and-parallel execution of non-deterministic goals. Thus, they are applicable to independent and-parallel systems such as &-Prolog [8] and &ACE [11], to dependent and-parallel systems such as DDAS [13], and to more general systems that incorporate and-parallelism such as Prometheus [13], ACE [7], and systems based on the Extended Andorra Model [15, **?**]. In this paper, however, we use an independent and-parallel implementation (*&ACE*) to illustrate the three optimizations. These three optimizations can lead to considerable efficiency of space and time. In many cases the optimizations can interact with each other and one optimization can enhance the effect of the other. In no case, however, the optimizations interact together to produce less efficient execution.

2 Independent And-parallelism

In this section we briefly describe independent and-parallelism and how it is implemented. Foundations of much of the work described in this section were laid down in [6], however, the specific implementation described is that of the &ACE system [11].

Conventionally, an and-parallel Prolog system works by executing a program that has been annotated with parallel conjunctions. These parallel conjunction annotations are either inserted by a parallelizing compiler [10] or hand-coded by the programmer. Execution of all goals in a parallel conjunction is started in parallel when control reaches that parallel conjunction. Whenever a parallel conjunction is met during execution, a data structure—the *parcall frame*—describing the parallel conjunction is allocated on the (control) stack. It contains various bookkeeping information (like number of subgoals in the conjunction), together with a descriptor—called *slot*—for each subgoal in the conjunction. At the same time, appropriate data structures (e.g. work queue)are initialized to allow remote execution of the newly generated subgoals.

Backtracking becomes complicated in and-parallel systems, because of the distributed nature of the computation and the possibility of concurrent backtracking activities along different branches of the execution tree. One of the most effective semantics for dealing with this sort of situation has been described my Hermenegildo and others [8]: the basic idea is to mimic Prolog-like backtracking (i.e., right-to-left analysis of subgoals) in the parallel execution (and this involves exchange of messages between processors in order to propagate backtracking and remove computations).

Independent and-parallelism with the backtracking semantics mentioned above has been implemented quite efficiently in the &ACE system [11]whose implementation is inspired by the RAPWAM [8][1]. The &ACE system has shown remarkable results on a variety of benchmarks. Its performance figures for the Sequent Symmetry multiprocessor can be found in [11].

3 Shallow Parallelism Optimization

During and-parallel execution a processor can pick up a goal for execution from other processors once it becomes idle. The general model of RAPWAM/&ACE requires at this stage the allocation of a data structure on the choice point stack, a *marker*. This marker is used to indicate the point at which the execution of a subgoal was initiated, partition the stack in *sections*, one for each parallel subgoal, and maintain the logical connections between a parallel subgoal and its parent computation. The same considerations need to be applied when a subgoal is completed—a marker needs to be allocated to indicate the completion of a subgoal and the end of the corresponding stack section.

The expense incurred in allocating these markers is considerable—markers are structures with as many as 10 fields that need to filled in at time of their allocation. Both the input and the end markers are needed mainly during backtracking. When an input maker is reached during backtracking it is an indication that no more backtracking is possible in the current and-parallel goal. In such a situation, backtracking is continued in the goal immediately to the left, starting from its end marker node. If it is known that a goal in a parallel conjunction is not going to produce any further solutions, i.e., it is *determinate*, then there is no need for keeping track of boundaries of this goal for backtracking. Thus, given a parallel conjunction

[1]Other implementations based on the principles of RAPWAM have also been proposed in the past, like &-Prolog [8] and DDAS [13].

$(\mathtt{g}_1\ \&\ \mathtt{g}_2\ \&\ \mathtt{g}_3)$[2], if it is known that $\mathtt{g}_2$ is determinate then backtracking should proceed directly from $\mathtt{g}_3$ to $\mathtt{g}_1$. We only have to make sure that every binding trailed during the execution of $\mathtt{g}_2$ is untrailed before backtracking moves into $\mathtt{g}_1$. This implies that actually there is no need to allocate the input marker node and the end marker node for goal $\mathtt{g}_2$ during forward execution.We term this optimization, where we avoid allocation of input and end marker nodes, *Shallow Parallelism Optimization*. The name is inspired by the similarity at the implementation level between this optimization and the *shallow backtracking* optimization originally introduced by Mats Carlsson [2]. To accomplish the shallow parallelism optimization, allocation of an input marker node is delayed until a goal is known to be non-determinate (i.e. until a choice point has been created) or until another parallel conjunction is encountered. When an and-parallel subgoal is picked up for execution from the goal stack, no input marker node is allocated in the control stack and the current top of the trail stack is recorded in the slot corresponding to this subgoal in the parcall frame. If, during execution, a choice point needs to be allocated, or another parallel conjunction is to be started, the (delayed) input marker node will be allocated first, and then the choice point node or the parcall node will be allocated space on the control stack next. If no choice point or parcall frame have been created during the execution of the goal and the end of the subgoal is reached, then the input marker node will *never* be allocated, saving time and space. At the same time the corresponding end marker will also not be created. The only additional operation required is the saving of the top of the trail stack in the slot at the end of execution of the subgoal—in this way the slot will keep track of the trail section used during the execution of the deterministic subgoal (needed during backtracking).

The shallow parallelism optimization is an illustration of an important optimization principle—*principle of procrastination* [9]—that has been ingeniously used over and over in the design of the Warren Abstract Machine (WAM). This principle states: *"An operation should be delayed until it is absolutely necessary to perform it"*. This is because in some cases it may turn out that the execution of this operation can be delayed forever (i.e. it does not need to be executed at all).

3.1 Experimental Results

The shallow parallelism optimization has been incorporated in the &ACE system. The results obtained have been extremely good. On average, an improvement of 5% to 25% over unoptimized implementation is obtained due to this optimization alone. In table 1 (note that all figures presented

[2]We use '&' to denote a parallel conjunction, while ',' is used to identify sequential conjunction.

Goals executed	&ACE agents			
	1	3	5	10
matrix_mult(30)	5.59/5.2 (7%)	1.9/1.7 (10%)	1.1/1.0 (8%)	.57/.53 (7%)
takeuchi(14)	2.4/1.8 (23%)	.83/.58 (30%)	.52/.36 (29%)	.25/.20 (21%)
hanoi(11)	2.2/1.6 (23%)	.76/.55 (28%)	.47/.33 (29%)	.23/.18 (22%)
poccur(5)	3.6/3.1 (12%)	1.2/1.0 (14%)	.75/.66 (13%)	.43/.37 (14%)
bt_cluster	1.4/1.3 (8%)	.52/.48 (9%)	.34/.31 (10%)	.20/.18 (7%)
annotator(5)	1.6/1.4 (12%)	.55/.47 (15%)	.39/.32 (18%)	.21/.18 (12%)

Table 1: Unopt./Opt. Exec. times in sec (% improvement)

in this paper are for the Sequent Symmetry multiprocessor) the execution times and relative percentage of improvement obtained on some common benchmarks are listed.
Observe that those benchmarks which show the best improvement under the Shallow Parallelism Optimization are those which contain considerable amount of parallelism (nesting of over 1000 parallel calls) and in which the "leaves" of the computation tree are deterministic computations (`hanoi` and `takeuchi` are two such benchmarks).

For other benchmarks the effects of the optimizations are more limited—for example in the matrix multiplication benchmark the whole computation is deterministic but the determinism is not detected because of the presence of nested deterministic parallel computations, and/or the presence of choice points whose remaining alternatives lead to failure. Furthermore, the optimization gives good results also on more complex benchmarks, involving backtracking across parallel subgoals. For example, running a program to solve a *map-coloring* problem (involving backtracking over parallel conjunctions), we obtained an average improvement in execution time of 14%. Clearly, since the main issue of this optimization is the avoidance of allocation of certain data structures, the computation will also gain considerable advantage in terms of memory consumption. Figure 1 illustrates the savings on the number of markers allocated obtained for some of the benchmarks. We go from an extreme case like `boyer` (no savings at all), to some excellent results, like for `takeuchi`, in which we save almost 50% of the total number of markers.

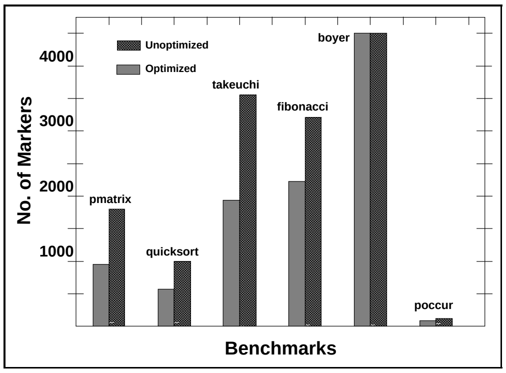

Figure 1: Number of markers allocated

4 Last Parallel Call Optimization

The LPCO illustrates two important optimization principles, namely: (i). *Reduced Nesting Principle*: The level of nesting of control structures in a computation should be reduced whenever possible; (ii). *Memory Reuse Principle:* Memory should be reused whenever possible.

The intent of the Last Parallel Call Optimization (LPCO) is to merge, whenever possible, distinct parallel conjunctions. 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 (and, eventually, triggering other optimizations—like the shallow parallelism one); *(ii)* It speeds up the process of backtracking; *(iii)* It saves space on the stacks and allows early recovery of space on backtracking. The advantages of LPCO are very similar to those for last call optimization [14] 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)` and `q :- (t & u)` are the clauses for `p` and `q`. 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 the query 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). 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. Note that this is only possible if `(p & q)` are determinate. The extra information added will consist of adding slots for the goals `r`, `s`, etc. Note that no new control information need to 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)`. It is also necessary to slightly modify the structure of a slot in order to adapt it to the new pattern of execution[3].

Note also that if the goal `r` is to fail in inside mode, then in case (ii) (see 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

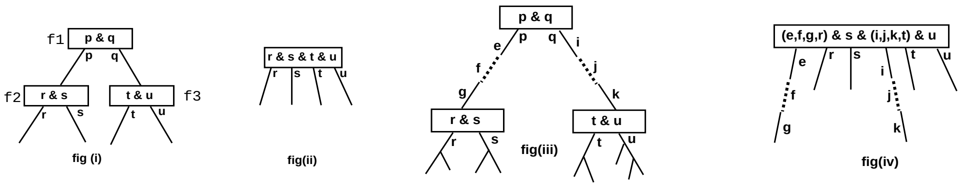

Figure 2: Reusing Parcall Frames

sent out to parcall frame f3.

[3]For example, it is necessary to keep in each slot a pointer to the environment in which the execution of the corresponding subgoal should start.

One could argue that the improved scheme described above can be accomplished simply through compile time transformations. However, in many cases this may not be possible. For example, if `p` and `q` are dynamic predicates or if there is not sufficient static information to detect the determinacy of `p` and `q`, then the compile-time analysis will not be able to detect the eventual applicability of the optimization. Our scheme will work even if `p` and `q` are dynamic or if determinacy information cannot be statically detected because trigerred only at runtime.

Also, for many programs the number of parallel conjunctions that can be combined into one will only be determined at run-time. For example, consider the following program:

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

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, since the the goal `process_list` is determinate, nesting of parcall frames can be completely eliminated. 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, with modest results (e.g., [3]).

Next we present the most general case of LPCO. This arises when there are goals preceding the parallel conjunction in a clause that matches a subgoal that is itself in a parallel conjunction (figure 2.(iii)). 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 *(i)* there is only one (remaining) matching clause for `p` (resp. `q`), i.e., `p` (resp. `q`) is determinate; *(ii)* 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, we can pretend as if the clause for `p` was defined as `p :- ((e,f,g,r) & s)` (although the bindings generated by `e, f, g` would be produced *before* starting the execution of `s`). Following the previous example, we extend the parcall frame for `(p & q)` with an appropriate number of slots and insert the nested parallel call in place of `p`. Likewise for the clause for `q`, if it contains a parallel call as its last call. This is illustrated in Figure 2(iv). 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` (resp. `q`) was not determinate in the beginning, the determinacy conditions will be satisfied when the last matching 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 the parallel subgoals in the clause for p (resp. q). Clearly, the goals r, s, etc. can be non-deterministic. When outside backtracking takes place in the tree in Figure 2(iv), because of the organization of the parcall frame, backtracking will proceed through u, t, i,j,k (without finding here any further solution since, by hypothesis, i,j,k must represent a deterministic computation) and so on. Backtracking over i,j,k will be immediate (since no choice points are present).

Suppose now an untried alternative is found within s, then the subgoals on the right of s have to be restarted. In this case the whole computation of p will be reactivated. This example shows one of the focal point in the implementation of the LPCO, the need to maintain a *backtrackable* description of the subgoals associated to a given parallel call. In the example above, once we have backtracked over i,j,k we need to *undo* the application of the LPCO, removing the new subgoals introduced (i,j,k, t, and u) and restoring the previously existing one (q). This can be avoided only if we have further evidence that actually only one clause (the one indicated above) matches subgoal q. Generalizations of the LPCO to more complex cases (like those when the clause used has a continuation, e.g. `p :- e, f, g, (r & s), h`) are not considered here due to lack of space.

4.1 Implementation of LPCO

To implement LPCO, the compiler will generate a different instruction whenever a parallel conjunct is encountered at the end of a clause. This instruction, named `opt_alloc_parcall`, behaves the same as the `alloc_parcall` instruction (the instruction used to create parallel conjunctions) of &ACE/RAPWAM, except that if the conditions for LPCO are fulfilled, then last parallel call optimization will be applied[4].

The introduction of the LPCO in the &ACE system requires only one related change in the architecture. In the original &ACE (as in RAPWAM, DDAS, etc.) the slots that are used to describe the subgoals of a parallel call are stored on the stack as part of the parcall frame itself. 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 as a linked list on the heap and a pointer to the beginning of the slot list stored in the parcall frame (Figure 3). The slot list can be maintained as a double linked list, simplifying the insertion/removal operations. Also, each input marker of an and-parallel goal has a pointer to its slot in the slot list for quick access (this is already part of the original &ACE design). Figure 3 illustrates this for the example

[4]The cost of the check is negligible.

in Figure 2(iv). Note that the modification of the slot list will have to be an atomic (backtrackable) 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 &ACE. The garbage collection mechanisms used on the heap guarantees that as soon as we have completely backtracked over a nested parallel call (optimized by LPCO) the space taken by the slots will be immediately recovered.

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, and any other operation on the slots requires anyway a linear scanning of all the slots in the parallel call.

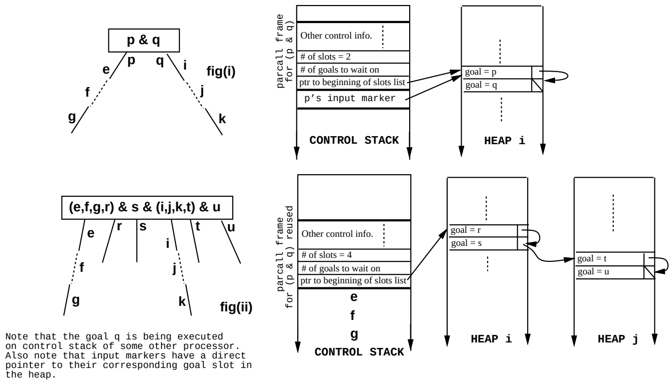

Figure 3: Allocating Goal Slots on the Heap

It is obvious that LPCO indeed leads to saving in space as well as time during parallel execution. In fact: (i) space is saved by avoiding allocation of the nested parcall frames; (ii) occasionally some time may be saved during forward execution (although the time complexity of applying LPCO is often comparable to the time complexity of allocating a parcall frame); and, (iii) time is saved during backtracking, since the number of control structure to traverse is considerably reduced.

4.2 Experimental Results

The LPCO optimization has been implemented as part of the current version of the &ACE and-parallel system. A first result that has to be mentioned is the limited amount of time and the limited amount of work required to add the LPCO to the existing implementation, thanks to its inherent simplicity. Introducing the LPCO took only a week of work and we strongly believe that porting it to different and-parallel systems will not require any larger effort. The experimental tests that we have performed consist of running various benchmarks, measuring the time elapsed and memory consumed during execution. In particular we selected the benchmarks in order to separately study the effects of the LPCO on programs whose execution: (i) is purely forward execution (i.e. no backtracking over parallel conjunctions); and, (ii) contains substantial backward execution (backtracking over parallel conjunctions).

Furthermore, we have separated our experimental analysis into two phases, by first running the benchmarks on the system with only LPCO and next executing them on the system with both LPCO and Shallow Parallelism Optimization. The two following subsections present the results obtained.

LPCO only

We use standard benchmarks that have been used by other researchers. For programs with only forward execution in parallel conjunctions (i.e., no backtracking across parallel executions) the results have not been extremely surprising: the cost of actually applying the LPCO is almost equivalent to the cost of creating a parcall frame. This is quite justifiable considering that most of the work during the creation of a parcall frame is spent initializing some basic fields of the frame (like the number of slots) and creating the slots for the various subgoals. Even with the LPCO both these operations still need to be performed, with the only difference that we are not initializing anymore the fields of a new structure but simply updating those of an existing parcall frame. The limited savings that we have using the LPCO are often lost due to the need of acquiring exclusive access to the parcall frame during application of the optimization. As a result, on most of benchmarks involving purely forward parallel execution we did not notice any difference in execution time. Only those benchmarks involving an extremely deep level of nesting of parallel calls show an improvement during forward execution. The timings obtained on a program which performs a naive search of an element in some nested lists (`search(1500)`) shows an improvement of 7% in execution time (going from 2222 *ms.* to 2077 *ms.*). Similar results can be expected for any other program with an analogous structure (deep recursion with independent iterations). It should be noted that LPCO, even for programs with purely forward execution, does not lead to slowdown.

Goals executed	&ACE Execution *Memory Required %*
bt_cluster	64%
pderiv	63%
pmatrix	67%

Figure 4: Fw. Execution

Goals executed	&ACE Execution	
	bw/no lpco	bw/lpco
Bt(0)	929	877 (6%)
Deriv(0)	151	120 (21%)
Occur(5)	3360	3255 (3%)
pmatrix(30)	6058	5800 (4%)
search(1500)	8370	3200 (62%)

Figure 5: Bw. execution

The results in terms of memory consumed are more interesting. For most benchmarks the number of parcall frames is cut down to one, i.e., only a main parcall frame is created and all the nested ones are removed. Table 4 compares the percentage of memory (i.e., percentage of control stack usage) required to execute some benchmarks using LPCO *vs* without using LPCO.

Considerable improvement in execution time is obtained for programs involving backward execution over parallel conjunctions[5]. The presence of the LPCO allows saving of execution time which is generally proportional to the

[5]Backtracking in many benchmarks is forced by adding a `fail` at the end of the query.

depth of the nesting of parallel calls. Some results are illustrated in table 5 (execution time in ms.). On some of the benchmarks the improvement is not very considerable (for example for the British Telecom clustering benchmark we have an improvement in execution time of 6%, and for the `occur` benchmark this goes down to 3%), due to the shallow nesting of parallel calls (50 or 60) and to the predominance of the actual computation time over parallel overhead. For other benchmarks (like the `search` and the `deriv`) the results are considerably better.

LPCO with Shallow Parallelism

One of the most interesting result that has emerged from interaction of LPCO with shallow parallelism optimization is that the use of LPCO considerably increases the applicability of the Shallow Parallelism Optimization. This is because the application of LPCO allows to flatten the nesting, exposing deterministic computations that otherwise would be hidden by the presence of other parallel calls nested inside. Table 2 summarizes the results obtained for various benchmarks for single processor execution. The table reports the results obtained for both purely forward and forward+backward execution, indicating within parenthesis the percentage of improvement obtained using LPCO with Shallow Parallelism Optimization.

Goals executed	&ACE Execution with Shallow Parallelism Optim.			
	fw/no lpco	fw/lpco	bw/no lpco	bw/lpco
Bt(0)	890	843 (5%)	929	853 (8%)
Deriv(0)	94	34 (64%)	131	38 (71%)
Occur(5)	3216	3063 (5%)	3352	3226 (4%)
pann(5)	1327	1282 (3%)	1334	1281 (4%)
pmatrix(20)	1724	1649 (4%)	1905	1696 (11%)
search(1500)	2354	1952 (17%)	8370	2154 (74%)

Table 2: Unoptim./Optim. Execution times in ms (single proc.)

The results are extremely good for programs with a certain structure. In particular, programs of the form `p(...) :- q(...) & p(...)`, where `q(...)` gives rise to a deterministic computation with a sufficiently deep level of recursion, will offer considerable improvement in performance. Interesting results are also seen by examining the effect of *inside failures* during execution: the use of LPCO allows further improvement. The presence of a single parcall frame considerably reduces the delay of propagating *Kill* signals. We tested the effects of the LPCO on a modified version of the matrix multiplication benchmark, in which an inside failure was forced at the deepest point of the recursion. The execution time, thanks to the composed effect of LPCO and Shallow parallelism optimization, results to be reduced from 5346 ms.

Goals executed	&ACE Execution	
	Markers/Parcalls	% Improvement
BTcluster	119/60 → 1/1	49 %
Deriv	134/87 → 0/1	45 %
Occur	95/50 → 40/1	34 %
Serial	16/11 → 0/1	43 %
Matrix	1638/1599 → 0/1	39 %

Table 3: Memory Usage (Unoptim. → Optim.)

to 3100 ms., with an improvement of 42%.

Also in terms of memory consumption, the combination of LPCO and Shallow Parallelism has proven to be extremely successful—while the LPCO cuts on the number of parcall frames the Shallow Parallelism removes the allocation of input and end markers. Table 3 summarizes these results: each entry of the form $a/b \rightarrow c/d$ indicates that the number of markers needed went down from a to c and the number of parcall frames went down from b to d when LPCO and shallow parallelism optimization were applied.

4.3 Comparison with other Works

At present we are not aware of any work which attempts to optimize last parallel call—at least at the level of control structures as we do. The works that come closest are:

- Ramkumar and Kalé's distributed last call optimization: designed for their ROPM system [12]. This optimization is specific to process based systems (like ROPM) and its main objective is to reduce the message flow between goals during parallel executions. The main idea is that, if a goal g calls a goal g' which in turns calls a goal g", then the solutions produced by g" can be directly communicated to g, if g" is the last call in g' and g' is the last call in g. In this way the communication through g' is avoided. The sole aim of the last distributed call optimization is to reduce message passing traffic in the multiprocessor system—so that its aim, scope, as well as its results are quite different from the traditional last call optimization or from our last parallel call optimization.
- Bounded quantification and reform execution of Prolog: LPCO can be seen as an instrument for taking advantage of occurrences of *Data Parallelism* in Prolog programs. Data Parallelism generally indicates a form of parallelism which can be exploited using the *Single Program Multiple Data* (SPMD) model—and its typical instance is represented by recursive clauses whose iterations can be performed in parallel. As observed in [3] data parallelism can be seen as a restricted form of and-parallelism. LPCO can be seen as a way to efficiently executing data parallel programs. Given a recursion like `p(...) :- q(...) & p(...)`, although a system like &ACE will produce one iteration at a time, the LPCO will actually collect all the iterations under a single parallel call—obtaining an effect analogous to a complete unfolding of the recursion. In particular this effect is present on backtracking. This compares to other proposals made in the literature for exploitation of data parallelism. The closest one is the work on Reform Prolog [1]. Reform Prolog's aim is to identify at compile time (through user annotations or compile-time analysis) the occurrences of data parallelism (like the recursive clause described above) and to generate specialized code, capable of efficiently unrolling the recursion and distributing the subgoals for parallel execution. This approach has advantage over

LPCO of being slightly more efficient for data-parallelism (since it is capable of unrolling in a single step the whole recursion).On the other hand, Reform Prolog exploits only a very specific form of parallelism (while LPCO can be mounted on top of a general and-parallel system), relying heavily on compile-time analysis. Furthermore it cannot deal with global non-determinism (i.e., non-determinism which spawns across different parallel computations). Comparing LPCO and Reform Prolog on some benchmarks we have observed comparable speedups, while Reform Prolog is on average 15% faster than &ACE with LPCO on sequential executions. On the other hand LPCO guarantees optimal saving in memory consumption and can be applied considerably more frequently than Reform Prolog.

- It should be noted that another notion of 'call optimization' is present in committed choice languages (like Parlog). This optimization is of quite a different nature from our LPCO: whenever a subgoal p commits to a certain clause, instead of spawning n new processes (one for each element of the body of the clause), it spawns only $n-1$ while one of the clause's subgoals (typically the last one) is automatically executed by the same process running p. Clearly the scope and aim of this optimization are different from those of LPCO.

5 Determinate Processor Optimization

The aim of a general and-parallel system is clearly to exploit the highest possible amount of parallelism, respecting the no-slowdown requirement. Nevertheless the amount of parallelism exploited is often greater than the actual amount of computing resources available—which leads to a situation in which the same computing resource (processor/agent/team of processors/etc.) will successively execute *different units* of parallel work, e.g. different subgoals of the same parallel execution. Thus, we get to a situation in which two potentially parallel pieces of computation are executed sequentially. The interesting situation occurs when the two units of works are actually executed in the same order in which they would be executed during a purely sequential computation—if this is the case then all the additional operations performed that are related to the management of parallel execution represent pure overhead. The intent of the *Determinate Processor Optimization (DPO)* is precisely to reduce this sort of overhead as much as possible.

DPO illustrates yet another principle of parallel programming. We term this principle the *Sequentiality Principle*: *"If two operations that can possibly be executed in parallel are at runtime executed by a single processor in the same order in which they would have been executed in sequential execution, then the parallel overhead for these two operations should be avoided as much as possible"*.
Clearly, as long as we rely on a dynamic scheduling mechanism (as &ACE,

RAPWAM, DDAS, etc. do), we don't really have any way of telling a priori when the mentioned condition will occur (unless some smart scheduling technique is adopted). As a consequence certain overhead cannot be avoided—like those related to the scheduling of the subgoals. On the other hand what we can do is the following: once a situation in which the optimization can be applied is detected—i.e. the scheduling returns (or, it manages to select) a subgoal which, considering sequential semantics, immediately follows the one just completed—we can avoid most of the overhead associated with the second subgoal. This saving is obtained by simply avoiding allocating any marker between the two subgoals and—in general—treating them as a unique, contiguous piece of computation (See Figure 6). There are several advantages in doing this: *(i).* memory consumption is reduced, since we are avoiding allocation of the markers between consecutive subgoals executed on the same processor; *(ii).* execution time during forward execution is reduced since the whole phase of creating an end marker for the first subgoal and an input marker for the second one (or a unique marker as happens in &ACE) is skipped; *(iii).* execution time during backward execution is also reduced, since backtracking on the two subgoals flows directly without the need of performing the various actions associated with backtracking over markers (sending messages, etc.).

From an implementation point of view, introducing DPO requires minimal changes to the architecture—at least to introduce the optimization in its simplest form. What is required is just an additional check on the exit of the scheduler to verify whether the new subgoal can be merged with the one previously executed. The check is immediate, at least in an architecture in which knowledge of the order of subgoals is maintained (like &ACE, RAPWAM, etc.). Improved results can be obtained by modifying the scheduler itself, forcing it to start its search by checking whether the immediately successive subgoal is still available for parallel execution.

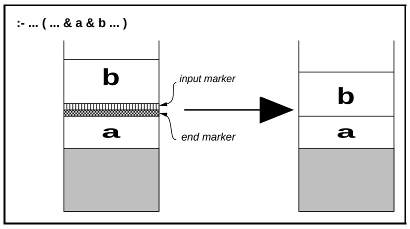

Figure 6: DPO

5.1 Experimental Results

The Determinate Processor Optimization has been included in the &ACE in its simplest form — i.e., only the check at the exit of the scheduler has been added, without changing the scheduler itself. We have tested the DPO over the same set of benchmarks used for the Shallow Parallelism optimization. Regarding forward execution, the results obtained are extremely encouraging, as can be observed from table 8. For many examples the optimization manages to improve the execution time from 3% to almost 20%. The variations in improvement depend exclusively on the number of parcall frames generated and on the effect of the marker allocation overhead on the overall execution time. For this reason we obtain considerable improvements in benchmarks like `takeuchi`, where we have deep nestings of parallel calls and the marker allocation represents the main component of the parallel overhead. The optimization maintains its effects when we consider execution spread across different processors, as we can see in figure 7, which shows the execution times of both the optimized and the unoptimized version for the Hanoi benchmark. The advantages are even more evident in terms of memory consumption: as we can see from table 9 the number of markers allocated is cut in almost all the cases to half of the original value—this because most of the examples analyzed have parallel calls of size 2 and the optimization allows to avoid the allocation of the marker between the two subgoals.

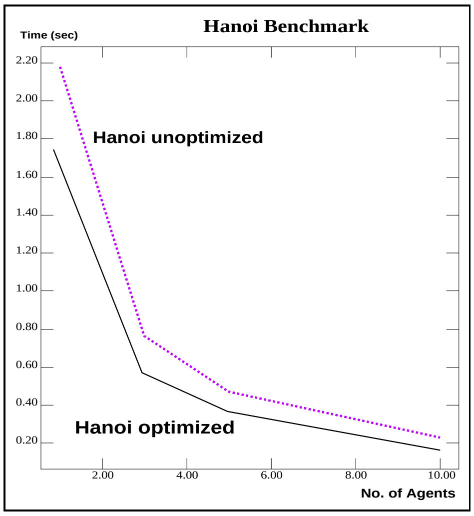

Figure 7: DPO: Hanoi Bench.

Goals executed	&ACE Execution Unopt.	Optim.
Bt(0)	1461	1391 (5%)
Occur(5)	3561	3418 (4%)
pmatrix(30)	5598	5336 (5%)
listsum	2333	2054 (12%)
hanoi	2183	1790 (18%)
takeuchi	2366	1963 (17%)

Figure 8: Execution times in ms

Goals executed	&ACE Execution Unopt.	Optim.
Bt(0)	120	60
Occur(5)	100	50
pmatrix(30)	1798	899
listsum	3000	1500
takeuchi	3558	2372
hanoi	4094	2047

Figure 9: Memory Consumptions

6 Conclusion

In this paper we presented three novel optimization called respectively Last Parallel Call Optimization, Shallow Parallelism Optimization, and Determinate Processor Optimization. These optimizations put well know optimizations principles into practice. The Shallow Parallelism optimization guarantees savings in memory usage and execution time whenever determinate subgoals are submitted for parallel execution. The Last Parallel Call optimization can be regarded as an extension of last call optimization, found in sequential systems, to and-parallel systems. Not only the LPCO saves space, it considerably speeds up backward execution and 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 are limited to management of the parcall frames. Finally, the Determinate Processor Optimization reduces the overhead whenever parallel execution mimics sequential execution. These three optimizations have been implemented in the &ACE parallel systems, a system currently under development in collaboration between New Mexico State University and the University of Madrid, and the experimental results confirm the effectiveness of these optimizations.

Acknowledgements

Thanks are due to Manuel Carro and Manuel Hermenegildo of University of Madrid and Kish Shen of the University of Bristol for many stimulating discussions. This research is supported by NSF Grant CCR 92-11732 and HRD 93-53271, Grant AI-1929 from Sandia National Labs, by an Oak Ridge Associated Universities Faculty Development Award, by NATO Grant CRG 921318 and by a Fellowship from Phillips Petroleum to Enrico Pontelli.

References

[1] J. Bevemyr, T. Lindgren, H. Millroth. Reform Prolog: the Language and its Implementation. In *Proc. Tenth Int'l Conf. on Logic Prog.*, MIT Press, 1993.

[2] M. Carlsson. On the Efficiency of Optimizing Shallow Backtracking in Compiled Prolog. In *Proc. Sixth Int'l Conf. on Logic Prog.*, MIT Press, 1989.

[3] M. Carro, M. Hermenegildo. A Note on Data-Parallelism and (And-parallel) Prolog. In *ICLP94 Post-conf. workshop on Parallel and Data Parallel Exec. of Logic Programs*, Uppsala University, 1994.

[4] V. Santos Costa, D.H.D. Warren, R. Yang. The Andorra-I Preprocessor: Supporting full Prolog on the Basic Andorra Model. In *ICLP*, MIT Press, 1991.

[5] S. K. Debray, D. S. Warren. Functional Computations in Logic Programs. In *ACM Transactions on Prog. Languages and Systems*, 11(3):451–481. 1989.

[6] D. DeGroot. Restricted AND-parallelism. In *Int'l Conf. on FGCS*, 1984.

[7] G.Gupta, E. Pontelli, M. Hermenegildo, V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *ICLP*, MIT Press, 1994.

[8] M. Hermenegildo, K.J. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *Proc. 7th ICLP*, MIT Press, 1990.

[9] D. Maier, D.S. Warren. *Computing with Logic: Logic Programming with Prolog.* Benjamin/Cummings Publishing Co., Inc., Menlo Park, CA, 1988.

[10] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *JLP*, 13(2 and 3), July 1992.

[11] E. Pontelli, G. Gupta, M. Hermenegildo. &ACE: A High-Performance Parallel Prolog System. In *Proc. IPPS'95.* IEEE Computer Society, 1995.

[12] B. Ramkumar. Distributed Last Call Optimization for Portable Parallel Logic Programming. In *ACM Letters on Prog. Languages and Systems*, 1(3), 1992.

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

[14] D. H. D. Warren. Last Call Optimization. "An Improved Prolog Implementation Which Optimises Tail Recursion," In *2nd ICLP*, 1984, Academic Press.

[15] D.H.D. Warren. The Extended Andorra Model with Implicit Control. *ICLP90 Pre-conference workshop*, June 1990.