Nested Parallel Call Optimization

Enrico Pontelli & Gopal Gupta*

Abstract

We present a novel optimization called *Last Parallel Call Optimization* (LPCO) for parallel systems. The last parallel call optimization can be regarded as a parallel extension of last call optimization (which itself is a generalization of the *tail recursion optimization*) found in sequential systems. While the LPCO is fairly general, we use and-parallel logic programming systems to illustrate it and to report its performance on multiprocessor systems. The last parallel call optimization leads to improved time and space performance for a majority of and-parallel programs. We also present a generalization of the Last Parallel Call Optimization called *Nested Parallel Call Optimization* (NPCO). The NPCO is also illustrated and its performance reported in the context of and-parallel logic programming systems. The LPCO and NPCO can be incorporated in a parallel system through relatively minor modifications to the runtime machinery; its incorporation in an existing and-parallel system is illustrated in this paper. A major advantage of LPCO and NPCO is that parallel systems designed for exploiting control parallelism can automatically exploit data parallelism efficiently. The LPCO and NPCO are motivated by two important principles of system optimization, namely, *the reduced nesting level principle* and the *memory reuse principle*.
Keywords: Implementation optimizations, Parallel logic programming, And-parallelism.

1 Introduction

Parallelization can add significant overhead to a system. Most of this overhead arises due to the extra book-keeping that has to be done to perform two ore more computations in parallel. This extra book-keeping adds an overhead in time as well as in space. Optimizing this book-keeping overhead (termed "parallel overhead") can result in improved system performance. Thus, optimizations that reduce the parallel overhead (both time and space) play a very important role in parallel implementations, since they lead to improved absolute performance. One can formulate a number of principles that an implementor of a parallel system should follow to minimize parallel time ands space overhead [10, 5]. Two such principles are:

- *Reduced Nesting Principle*: The level of nesting of control structures in a computation should be reduced whenever possible.
- *Memory Reuse Principle:* Memory should be reused whenever possible.

The usefulness of the memory reuse optimization principle is obvious. The application of the reduced nesting principle can be intuitively seen as follows: suppose in a parallel program, n parallel computations are spawned. Typically, some sort of descriptor data-structure will have to be allocated to keep track of the status of these n parallel computations. Suppose one of the parallel task further spawns m parallel subcomputations. A new descriptor will have to be allocated to keep track of these m parallel tasks. The principle of reduced nesting states that it is better to treat this computation as consisting of $m + n$ parallel tasks for which a single descriptor is allocated rather than as consisting of a nested parallel computation which requires allocation of two (or more, if other tasks also lead to parallel subcomputations) descriptors. Of course, while carrying out this merging of descriptors one has to ensure that semantics of the parallel computation are not altered.

In this paper we present the *Last Parallel Call Optimization* that is inspired by the above two principles. LPCO leads to considerable reductions in parallel overhead both in terms of time and space. We consider LPCO in the context of *And-parallel Prolog* systems where such nested parallel computation arise very frequently. Parallel execution in Prolog is more complex due to presence of backtracking. As a result, development of optimizations is also quite more involved.

Logic programming is a paradigm of programming based on a subset of Horn Logic. A distinguishing fea-

*Laboratory for Logic, Database, and Advanced Programming, Department of Computer Science, New Mexico State University, Las Cruces, NM, USA. {epontell,gupta}@cs.nmsu.edu

ture 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 parallel.

(ii). *And-parallelism:* arises when a set of conjunctive goals in the current gola-list 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 (termed *independent and-parallelism*), or they could be *dependent* in which case they will be executed in parallel until they access the common variables (termed *dependent and-parallelism*).

We present LPCO and NPCO for systems exploiting independent and-parallelism although our results are also applicable to dependent and-parallel systems and to 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 [6, 7, 12] and its efficient implementation in RAPWAM [7]. The backtracking semantics as given by Hermenegildo and Nasr attempts to emulate the backward execution control of Prolog as much as possible.

The Last Parallel Call Optimization is triggered when the last call in a Prolog clause is itself a parallel conjunction (from now on a parallel conjunction will also be referred to as a *parcall* for brevity). The Nested Parallel Call Optimization is triggered when a Prolog clause has a nested parallel conjunction, and all goals following the parallel conjunction (these goals are termed *continuation* of the parcall) satisfy certain conditions. The LPCO and NPCO, when applicable, simplify backtracking and allow failures and *kills* to be propagated faster. We present some experimental results to demonstrate their advantages. The LPCO and NPCO are parallel analogs of the last call optimization[1] that was designed by D.H.D. Warren for sequential Prolog execution [13]. When applicable, LPCO can speed up the execution of parallel goals considerably, while also resulting in a dramatical reduction in memory consumption.

The LPCO and NPCO are very general optimizations that are applicable to independent and-parallel systems such as &-Prolog [7] and &ACE [9], to dependent and-parallel systems such as DDAS [12], and to more general systems that incorporate and-parallelism such as Prometheus [12] and ACE [4]. They can also be applied to other languages, such as to parallel implementations of functional languages and of Fortran, C, etc. In this paper, however, we use an independent and-parallel implementation (named *&ACE* running on a Sequent Symmetry Multiprocessor) to illustrate LPCO and NPCO.

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, 3], however, the specific implementation described is that of the &ACE system [9].

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 [7] 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 encountered during execution, a descriptor 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 structure—called *slot*—that contains control information for each subgoal in the parallel conjunction. At the same time appropriate data structures (e.g. work queue, termed

[1] Last call optimization itself is a generalization of the tail recursion optimization.

goal stack in the logic programming literature) are initialized to allow remote execution of the newly generated subgoals (see Figure 1).

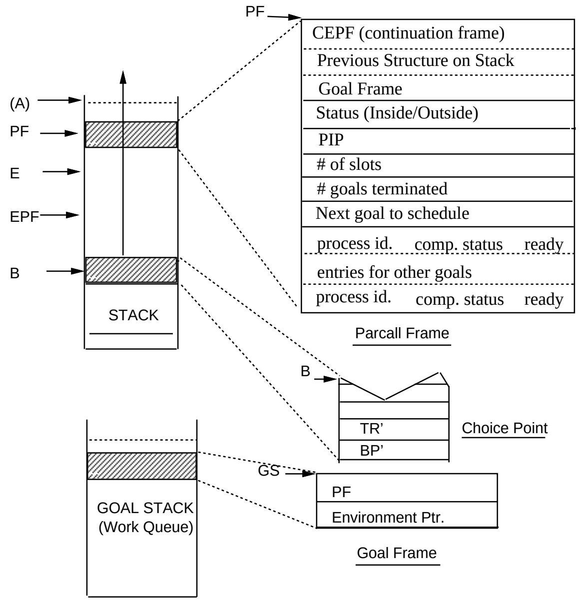

Figure 1: Datastructures needed for and-parallelism

A processor taking a subgoal from a parallel conjunction will initially allocate an *input marker* on its stack, to identify the beginning of the section of computation dedicated to this subgoal, and start the corresponding computation. At the completion of the subgoal another marker, the *end marker*, will be allocated to close the section[2].

Backtracking becomes complicated in and-parallel systems 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, and the distributed nature of the execution may require a considerable synchronization activity between the different computing agents. 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 [7]: 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 ("Prolog-like") occurs, as usual. Since `c`, `d`, and `e` are mutually independent, if any one of them fails before finding any solution, then a limited form of intelligent backtracking can be applied and backtracking can be forced to continue directly from `b` after *killing* the other sibling parallel computations inside this parcall. 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 (following Prolog semantics). 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*). When backtracking is initiated from outside, once a choicepoint is found in a subgoal p, an untried alternative is picked from it and then all the subgoals to the right of p in the parallel conjunction are restarted (in parallel).

Independent and-parallelism with the backtracking semantics described above has been implemented quite efficiently by the authors in the &ACE system [9]. &ACE implementation itself is inspired by the RAPWAM [7][3]. &ACE is an extension to the sequential WAM (Warren Abstract Machine [14]) for and-parallel execution of Prolog programs with and-parallel annotation. The &ACE system has shown remarkable results on a variety of benchmarks. Its performance figures for the Sequent Symmetry multiprocessor can be found in [9].

3 L. P. C. O.

The intent of the Last Parallel Call Optimization (LPCO) is to merge, whenever possible, distinct parallel conjunctions. Last Parallel Call Optimization can lead to a number of advantages (discussed later). The advantages of

[2] Pointers to the input and end markers are stored in the slot of the corresponding subgoal in the parcall frame for backtracking purposes.

[3] Other implementations based on the principles of RAPWAM have also been proposed in the past, like &-Prolog [7] and DDAS [12].

LPCO are very similar to those for last call optimization [13] 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 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 [6] 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. 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. In particular, 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[4].

Furthermore, 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 sent out to parcall frame f3. In case (ii) a linear scan of the goal list is sufficient.

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 it is triggered 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([], []).

?-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, since the the goal `process_list` 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. Hermenegildo and others [2] have suggested to partially unfold at compile-time 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 ($n = 3$). Nevertheless these kind of approaches have some obvious limitations (determining the best degree of unfolding, etc.).

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

......
```

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

[4] 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.

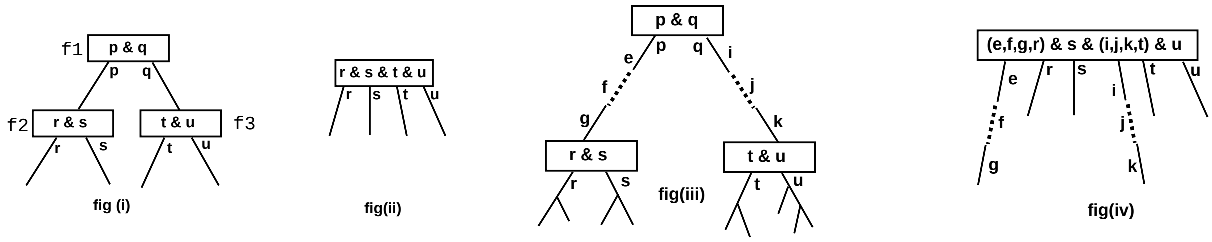

Figure 2: Reusing Parcall Frames

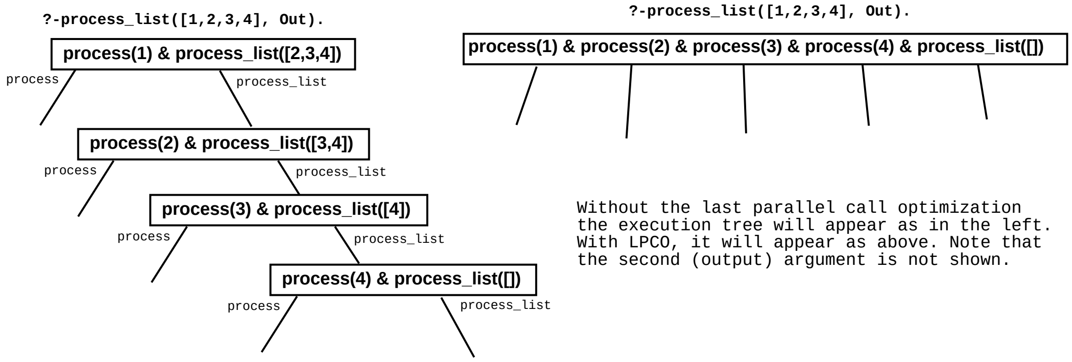

Figure 3: Reuse of Parcall Frames for Recursive Programs

in a parallel conjunction (figure 2(iii)). Thus, given a parallel conjunctions 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).

The two determinacy conditions above can be more easily understood by looking at the layout of the information on the control stacks: conditions *(i)* and *(ii)* are equivalent to the non-existence of any choice point between the current parallel call (e.g., the parcall `(r & s)`) and the logically preceding one (e.g., the parcall `(p & q)`). 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 for LPCO to apply) and so on. Backtracking over `i,j,k` will be immediate (since no choice points are present); the presence of a slot in the parallel computation descriptor dedicated to this deterministic part of the execution may seem superfluous, but it is actually necessary in order to guarantee a proper unwinding of the bindings created (and, in addition, it simplifies the management of the local stack). This can be improved further, but we omit the description of these improvements due to lack of space.

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[5]. This examples shows one of the important 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`.

3.1 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. In &ACE, the compiler (based on RAPWAM) generates an `alloc_parcall` instruction that places the parcall frame on the stack. To implement LPCO, a new instruction is generated instead. This instruction, named `opt_alloc_parcall`, behaves the same as the `alloc_parcall` instruction of &ACE except that if the conditions for LPCO are fulfilled the last parallel call optimization will be applied.

The behaviour of `opt_alloc_parcall` at runtime is quite straightforward: once it is reached, it will perform a check to verify whether the conditions for the application of LPCO are met (i.e. no choice points are present between the current point of execution and the immediate ancestor parcall frame). If the check succeeds, then LPCO will be applied, otherwise the instruction will proceed creating a new parcall frame (i.e., behaving like a normal `alloc_parcall` instruction). The above mentioned check is immediate and introduces insignificant overhead—it is sufficient to check that the data structure currently lying on the top of the stack is a parcall frame or an input marker.

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).

Introducing 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 on the heap and a pointer to the beginning of the slot list stored in the parcall frame (Figure 4). 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). With the linked list organization, adding new slots becomes quite simple. Figure 4 illustrates this for the example 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 a linear scanning of all the slots in the parallel call.

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) time is saved during forward execution (although for many programs the time complexity of applying LPCO

[5] Note that while we illustrate the optimization in the context of the backtracking scheme of Hermenegildo and Nasr, the LPCO will apply even if other backtracking semantics are adopted.

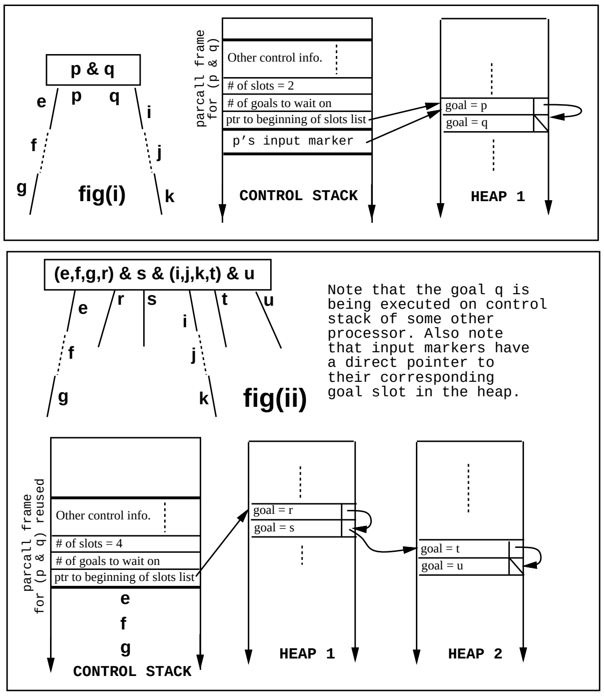

Figure 4: Allocating Goal Slots on the Heap

is often comparable to the time complexity of allocating a parcall frame); and, (iii) considerable time is always saved during parallel backtracking and unwinding of the stack[6], since the number of control structure to traverse is considerably reduced. Note that LPCO maintains its effect even when more and more processors are added for parallel execution.

4 Nested Parallel Call Opt.

As we mentioned in the previous sections, LPCO can be applied whenever certain conditions on the determinacy of given parts of the computations are met. The obvious question that comes to mind is whether these conditions can be relaxed and what eventually would be the cost of this relaxation. Two different extensions are discussed below.

Nondeterministic Computations: LPCO cannot be applied whenever a nondeterministic computation is performed between the two nested parallel calls, e.g. in `(p & q)` given a clause `p :- e , (f & g)` where `e` has multiple solutions. Extending LPCO to these cases is possible but it requires more involved changes in the slot list management and in the way in which backtracking is performed. This applies mainly to inside backtracking: if we apply the optimization to the case above we will obtain an *expanded* parallel call `( (e, (f & g)) & q )` where different slots of the parcall frame will be assigned to `e`, `f`, `g`, and `q` (figure 5(i)).

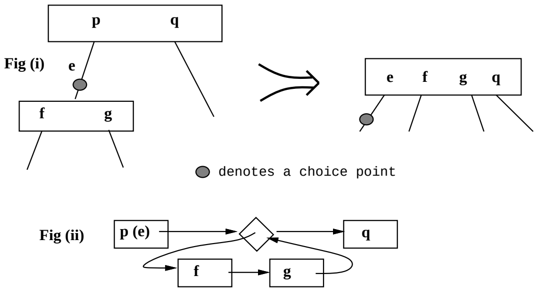

Figure 5: LPCO and Nondeterminate Computations

If the execution of `f` fails we are not allowed to apply standard inside backtracking and abort the whole parallel call (`q` included), since there could be other solutions for `e` which may cure the failure.

Nevertheless, a more sophisticated representation of the list of goals (in which the information about the nesting of parallel conjunctions is not completely lost) and a slight improvement in the backtracking mechanism are sufficient to allow extension of the LPCO to these cases. The key is to keep track of the points in which the optimization has been applied (replacing an existing goal with a new sublist of goals) as shown in figure 5(ii). Inside backtracking on this new structure will work as follows:

- ◇ standard inside backtracking is applied to the subgoals belonging to the same level of nesting (i.e., in the example respectively `p`, `q` and `f`, `g` are on the same level of nesting);
- ◇ backtracking is eventually propagated to the higher level of nesting by causing a failure to the immediately preceding subgoal in the list (i.e., if `f` fails, the failure will be propagated to `p`).

Continuations: LPCO cannot be applied whenever a computation is present in the continuation of the nested parallel calls. Let us consider a generic situation: a goal of the form `:- (p & q), c.` is solved against the clauses `p :- (`p_1` & `p_2`), `c_1`.` and `q :- (`q_1` & `q_2`), `c_2`.` Note that the nested parcalls are not at the end of the clause. The reason LPCO cannot be applied in this case is re-

[6] In functional languages and conventional languages there is no backtracking, however, the descriptors stored in the stack have to be removed at the end of the parallel computation. Reducing the level of nesting of parcalls is going to make space reclamation from the stack faster when parallel tasks are completed.

lated to respecting the desired order of execution: when the clause `p :- (p1 & p2), c1` is used in solving `(p & q)` and nesting of parcall is collapsed, we need to make sure that the execution of c_1 is not started before both p_1 and p_2 have completed. A safe possibility is to delay all the continuations of the nested parallel calls until the main parcall has completed. In the example, this equates to executing the goal $(p_1$ & p_2 & q_1 & $q_2)$, c_1, c_2, `c`. The soundness of this solution is guaranteed by the independence of the subgoals, which allows to delay the execution of the continuations of the nested parallel calls. Nevertheless, in order to have soundness we must also guarantee that the continuations, even if delayed, are executed in the proper order (i.e., if a subgoal `b` is expanded with a clause containing `(c & d), h` and `d` is expanded with `(e & f), i`, then `h` cannot be executed before `i`). This order of execution can be guaranteed by using a LIFO structure in which the continuations are pushed whenever parcalls are merged. Once the parallel conjunction is finished, the accumulated continuation goals can be picked from the stack one after the other and executed. However, this technique has the drawback that execution of all continuation goals becomes sequential, when some of them actually can be executed in parallel. This can be rectified by having an appropriate data-structure instead of a LIFO stack to record the continuation goals, but we will not describe it due to lack of space.

The above problem with continuation goals arises if continuations are non-deterministic, or if they fail causing failure from outside in the corresponding parallel conjunctions. If it is known that the continuation is deterministic, and is non-failing, then the continuation goals of a parcall can be executed without regard to continuation goals of other parcalls. In practice, it turns out that for most parcalls, either the continuation is empty, or it contains deterministic, non-failing goals. This extension of the LPCO has been actually implemented in the current version of the &ACE system (and some experimental results are reported in section 5).

Note that relaxing the conditions imposed on LPCO makes it a more general optimization scheme and increasing its applicability to a wide family of computations involving nested parallel calls. For this reason we term the generalization of LPCO the *Nested Parallel Call Optimization* (*NPCO*).

5 Experimental Results

The LPCO optimization has been implemented as part of the current version of the &ACE and-parallel system running on a Sequent Symmetry [9]. Introducing the LPCO took only a week of work, thanks to its inherent simplicity, and we strongly believe that porting it to different 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. 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 parcalls); and, (ii) contains substantial backward execution (backtracking over parcall).

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/NPCO and other optimizations [10]. A nice property of the LPCO and NPCO is that they flatten the nesting of parallel calls, as a result of which a number of other optimizations become applicable. The two following subsections present the results obtained.

LPCO: The execution of the system with the use of LPCO produces considerable speed-ups while maintaining a good efficiency in execution; figure 6 shows the speed-up curves obtained on some commonly used benchmarks (like `Takeuchi`—which computes a complex recursive functions, and `BT_Cluster`—an extract of a clustering program used by British Telecom) and on a "real-life" application, the full PLM Prolog compiler (`Compiler`) from U.C. Berkeley.

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

Goals executed	&ACE Execution			
	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 1: Unoptimized/Optimized Execution times in msec (single processor)

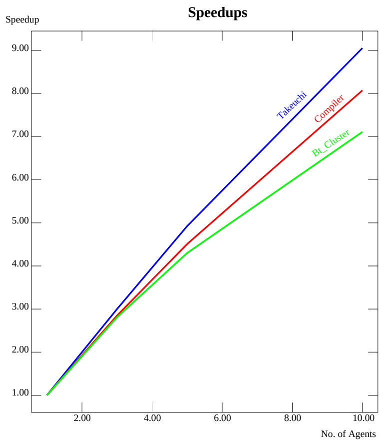

Figure 6: Speedups of &ACE with LPCO

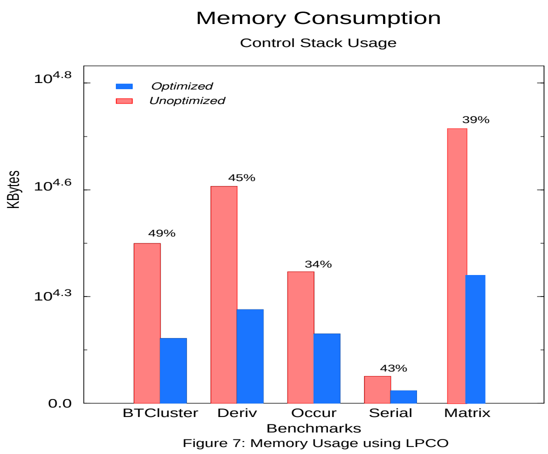

Figure 7: Memory Usage using LPCO

of LPCO allows further improvement. The presence of a single parcall frame considerably reduces the delay of propagating Kill signals to sibling parallel goals. In programs with sufficient nesting of parcalls, the improvement in total execution time due to faster killing improves by as much as 42%. Figure 7 summarizes memory savings obtained by LPCO: the picture compares the usage of control stack measured during the execution of some benchmarks, comparing the unoptimized vs. the optimized case. The percentages indicated show the reduction in memory consumption obtained.

NPCO: In the present version of &ACE only the second extension described in section 4 has been implemented. This version of NPCO has been tested on several benchmarks and the general results obtained are consistent with those presented in the previous section for LPCO: a moderate improvement in execution time for purely deterministic executions, a more considerable speed-up for computations involving backtracking across parallel executions, and, in general, a dramatic improvement in memory usage. Table 2 shows the results obtained for two benchmarks, `hanoi` and `quicksort` (both cannot take advantage of LPCO since the parallel call is not the last call in the clause)[7].

6 Data-parallel Programming

LPCO and NPCO can be seen as instruments for taking advantage of occurrences of *Data Parallelism* in Prolog programs. Typical instances of data-parallelism are represented by recursive clauses whose iterations can be performed in parallel. For example, the `process_list` program in section 3 is a data-parallel program, because once the recursion is completely unfolded a number of identical looking calls for the `process` goal are produced. As observed in [2] 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

[7] The results indicates the time employed for searching *all* the solutions to the original query—this justifies the low speedups caused by high overhead due to backtracking. Also, a 16MB memory board in our Sequent Symmetry broke down while collecting the data for NPCO, so the speed-ups are also low due to thrashing that is caused by shortage of physical memory. We hope to have the precise results in the final paper, if accepted.

Goals executed	&ACE agents 1	3	5	Memory Usage
quicksort	1848/1675 (9%)	993/699 (20%)	818/678 (17%)	223/150 (33%)
hanoi	3925/2563 (35%)	2359/1747 (26%)	2160/1588 (26%)	949/614 (35%)

Table 2: Execution under NPCO (execution time (ms.) and memory consumption (KBytes)

complete unfolding of the recursion. The efficiency of execution of data-parallel programs using LPCO and NPCO compares favorably to other proposals made in the literature for exploitation of data parallelism. The closest one (among many proposals made) 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: (i) performing all the head unifications required by the iterations of the recursion at once; and, (ii) completely unrolling the recursion at runtime and distributing the different iterations to different processors. Reform Prolog, due to its specialized nature, is slightly more efficient than LPCO/NPCO for data-parallel programs. If the depth of the recursion is n, Reform Prolog manages to unroll the whole recursion in a single step, while LPCO requires n steps to produce all the parallel work for parallel execution). On the other hand:

- Reform Prolog exploits only a very specific form of parallelism, while LPCO/NPCO can be mounted on top of a general and-parallel system;
- Reform Prolog relies heavily on compile-time analysis, while LPCO/NPCO is a purely runtime technique;
- Reform Prolog cannot deal with global non-determinism (i.e., non-determinism that spreads across different parallel computations);

Comparing LPCO and Reform Prolog on some benchmarks we have observed comparable speedups, while Reform Prolog is on average 10%-15% faster than &ACE with LPCO/NPCO on sequential executions. On the other hand LPCO guarantees optimal saving in memory consumption and can be applied considerably more frequently than Reform Prolog, i.e., LPCO applies even when a program may not be data-parallel in nature.

7 Conclusions

In this paper we presented two novel optimizations called Last Parallel Call Optimization and Nested Parallel Call Optimization. These optimizations put well known optimization principles into practice. 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 and NPCO save space, they considerably speed up the execution of a majority of 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. These optimizations have been implemented in the &ACE parallel systems, a system collaboratively being developed by New Mexico State University and the University of Madrid, and the experimental results confirm the effectiveness of these optimizations. These optimizations were illustrated in the context of and-parallel logic programming system, but they are applicably to any arbitrary parallel system that has nested parallel computation. Thus, LPCO and NPCO can be applied in parallel implementations of functional languages, as well as those of Fortran or C. The LPCO and NPCO illustrate two important principles of optimization: the reduced nesting principle and the memory reuse principle.

At present we are not aware on any work which attempts to optimize nested parallel calls—at least at the level of control structures as we do. One the works that come closest is Ramkumar and Kalé's distributed last call optimization designed for their ROPM system [11]. 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 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/nested parallel call optimization. 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 also different from those of LPCO/NPCO.

8 Acknowledgements

Thanks are due to Manuel Carro and Manuel Hermenegildo of University of Madrid, to Kish Shen of the Universities of Bristol and Manchester, and to Dongxing Tang of NMSU 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 International Conference on Logic Programming*, MIT Press, June 1993, pages 283–298.

[2] M. Carro, M. Hermenegildo. A Note on Data-Parallelism and (And-parallel) Prolog. In *Proc. of the ICLP94 Post-conference workshop on Parallel and Data Parallel Execution of Logic Programs.*

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

[4] G. Gupta, E. Pontelli, M. Hermenegildo, V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proc. of the Eleventh International Conference on Logic Programming*, MIT Press, June 1994, pages 93-109.

[5] G. Gupta, E. Pontelli. Optimizing Principles for Parallel Non-deterministic Systems. Technical Report, Dept. of Computer Science, NMSU, December 1994.

[6] M. Hermenegildo. An Abstract Machine for Restricted And Parallel Execution of Logic Programs. In *Proc. of the Third International Conference on Logic Programming*, 1986.

[7] M. Hermenegildo, K.J. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *Proc. of the Seventh International Conference on Logic Programming*, MIT Press, June 1990, pages 253-268.

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

[9] E. Pontelli, G. Gupta, M. Hermenegildo. &ACE: The And-Parallel Component of ACE (A progress Report). In *Proc. of the ICLP94 Post-conference workshop on Parallel and Data Parallel Execution of Logic Programs*, June 1994. Submitted to International Parallel Processing Symposium.

[10] E. Pontelli, G. Gupta, D. Tang. Determinacy Driven Optimization of Parallel Prolog Implementations. Technical Report, Dept. Computer Science, NMSU, November 1994.

[11] B. Ramkumar. Distributed Last Call Optimization for Portable Parallel Logic Programming. In *ACM Letters on Programming Languages and Systems*, 1(3), September 1992, pages 266-283.

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

[13] 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.

[14] D.H.D. Warren. An Abstract Instruction Set for Prolog. Tech. Note 309, SRI International, 1983.