(Proc. 1989 Int'l. Conf. on Logic Programming, MIT Press)

Complete and Efficient Methods for Supporting Side-effects in Independent/Restricted And-parallelism

K. Muthukumar
M. Hermenegildo

MCC and University of Texas at Austin, C.S. Dept.
Net Address: muthu@cs.utexas.edu, herme@cs.utexas.edu

Abstract

It has been shown that it is possible to exploit Independent/Restricted And-parallelism in logic programs while retaining the conventional "don't know" semantics of such programs. In particular, it is possible to parallelize pure Prolog programs while maintaining the semantics of the language. However, when builtin *side-effects* (such as `write` or `assert`) appear in the program, if an identical observable behaviour to that of sequential Prolog implementations is to be preserved, such side-effects have to be properly sequenced. Previously proposed solutions to this problem are either incomplete (lacking, for example, backtracking semantics) or they force sequentialization of significant portions of the execution graph which could otherwise run in parallel. In this paper a series of side-effect synchronization methods are proposed which incur lower overhead and allow more parallelism than those previously proposed. Most importantly, and unlike previous proposals, they have well-defined backward execution behaviour and require only a small modification to a given (And-parallel) Prolog implementation.

1 Introduction

It has been shown [10, 9] that it is possible to exploit Independent And-parallelism (where only sets of goals which do not share variables can be executed in parallel – [4], [3], [11], [2], [7], [12], [16], [17], ...) while retaining the conventional "don't know" semantics of such programs. In particular, it is possible to parallelize pure Prolog programs while maintaining the full syntax and semantics of the language. However, when builtin *side-effects* (such as `write` or `assert`) appear in the program, if an identical behaviour to that of sequential Prolog implementations is to be preserved, such side-effects have to be properly sequenced. For example, consider the following clause

```
p(X,Y) :- a(X), write(X), b(X,Y), write(Y).
```

and assume that `X` and `Y` are always bound to ground terms upon entry into `p`. In this case all subgoals in the body of `p` are independent and could in principle be executed in parallel.

However, because of the presence of side-effects, the four subgoals cannot be truly executed in parallel or in any arbitrary order if a behavior identical to that of a sequential execution is to be preserved: `a(X)` and `b(X,Y)` can be first executed in parallel while the execution of `write(X)` should be delayed until the execution of `a(X)` is over. This is necessary to avoid `X` being written in case `a(X)` fails. Similarly, the subgoal `write(Y)` should wait for the completion of execution of subgoals `b(X,Y)` and `write(X)`. The same argument will hold if `write(X)` and `write(Y)` are replaced by subgoals `a1(X)` and `b1(Y)` which are 'side-effect procedures' (i.e. they contain a call to a side-effect within the subtree that they generate).

One simple solution, for example, chosen in PEPSys [16], is to separate the program into sequential and parallel modules, with side-effects only being allowed in sequential ones. This solution is correct although it can potentially force sequentialization of significant portions of the program which could otherwise run in parallel. Some other solutions have been proposed within the scope of Or-parallelism [6], generally based on a run-time exploration of the execution tree to the left of a given side-effect. A side-effect synchronization method along these lines is proposed as an alternative in [13].

If side-effects are allowed within parallel code and a behaviour of the program identical to that observable on a sequential implementation is to be preserved then a good solution appears to be to add some sort of synchronization code to the program. In [5], DeGroot proposed a solution along such lines, based on the use of "synchronization blocks." DeGroot's approach is (at least during forward execution) correct and achieves its objective. However, it also suffers from the following limitations:

- First, and most importantly, as mentioned by DeGroot, the model is incomplete because it has no backtracking semantics, i.e. it only solves the side-effect synchronization problem if no goal fails during the execution of the program.

- The method is based on the use of synchronization blocks, which are complex data structures that are external to the standard Prolog storage model. This complicates the implementation so that relatively significant modifications to the parallel abstract machine may be required.

- DeGroot's method is only applied to the case when all goals in the program are executed in parallel. The case where there is a mixture of sequential and parallel goals (and, more generally, the case where sets of goals can conditionally be executed either sequentially or in parallel) needs to be addressed.

- Also, in DeGroot's method all side-effects are synchronized by a single

chain of synchronization blocks (semaphores). This limits parallelism unnecessarily.

- Finally, DeGroot's method doesn't make any provision for "parallel builtins."

In this paper we present a new solution to the side-effect synchronization problem (and we also mention two other alternatives). This solution is based on ideas similar to DeGroot's but solves all the shortcomings pointed out above.

The rest of the paper is organized as follows. Sections 2.1 and 2.2 introduce some annotation syntax and terminology. Section 2.3 then describes a desirable behavior of an And-parallel system in the presence of side-effects, and presents our technique for synchronizing side-effects in order to achieve such behaviour. Section 3 shows how the model operates during backtracking and describes the abstract machine modifications needed to implement such operation. Section 4 then describes some optimizations which reduce the number of semaphores needed and increase the attainable parallelism. Finally, section 5 presents two alternative implementation methods, and section 6 summarizes our conclusions.

2 A Backtrackable Method for Synchronizing Side-Effects

2.1 &-Prolog and the RAP-WAM Model: An Annotation for Parallelism

We believe that the techniques presented herein are applicable to most models based on Independent And-parallelism. However, for the sake of concreteness, the discussion will be presented in terms of the Independent/Restricted And-parallel (RAP-WAM) model [7, 8]. This model has its roots in DeGroot's seminal work on *Restricted And-Parallelism* [4]. RAP-WAM completes DeGroot's RAP by providing backward execution semantics to the model, improved graph expressions (&-Prolog's CGEs)[7],[1] and an efficient implementation model based on the Warren Abstract Machine (WAM) [14]. The RAP-WAM model is used in this paper not only for the sake of concreteness but also because of the convenience of its Prolog-compatible source language, &-Prolog, which makes it possible to discuss the side-effect synchronization techniques directly on the source program and to describe and implement the algorithms for adding side-effect synchronization code as a series of rewritings of the original Prolog program.

&-Prolog is basically Prolog, with the addition of the parallel conjunction operator "`&`" and a set of parallelism-related builtins, which includes several

[1] &-Prolog's CGEs offer Prolog syntax and permit conjunctive checks, thus overcoming the difficulty in expressing 'sufficient' conditions for independence with the expressions proposed by DeGroot: given "`f(X,Y,Z):- g(X,Y), h(Y,Z).`" the most natural annotation for this clause, that `g` and `h` can run in parallel if the terms in `X` and `Z` don't share variables and `Y` is bound to a ground term, can be expressed easily with CGEs ("`f(X,Y,Z):- (indep(X,Z), ground(Y) => g(X,Y) & h(Y,Z) ).`") but is very difficult with DeGroot's expressions.

types of groundness and independence checks, and synchronization primitives. Parallel conditional execution graphs (which cause the execution of goals in parallel if certain conditions are met) can be constructed by combining these elements with the normal Prolog constructs, such as "`->`" (if-then-else). For syntactic convenience (and historical reasons), an additional construct is also provided: the Conditional Graph Expression (CGE). A CGE has the general form

`(` *i_cond* `=>` $goal_1$ `&` $goal_2$ `&` `...` `&` $goal_N$ `)`

where the $goal_i$ are either normal Prolog goals or other CGEs, and *i_cond* is a condition which, if satisfied, guarantees the mutual independence of the $goal_i$s. The CGE can be viewed simply as syntactic sugar for

`(` *i_cond* `->` $goal_1$ `&` $goal_2$ `&` `...` `&` $goal_N$
`;` $goal_1$ `,` $goal_2$ `,` `...` `,` $goal_N$ `)`

i.e., the operational meaning of the CGE is "check *i_cond*, if it succeeds execute the $goal_i$ in parallel, else execute them sequentially." The $goal_i$ can themselves be CGEs, i.e. CGEs can be nested in order to create more complex execution graphs. *i_cond* can in principle be any set of Prolog goals but is in general either true (an "unconditional CGE") or a conjunction of checks on the groundness or independence of variables appearing in the $goal_i$s. Thus, the following &-Prolog clause

```
p(X,Y) :- q(X,Y), r(X), s(X).
```

could perhaps be rewritten for parallel execution as

```
p(X,Y) :- (ground(X) => q(X,Y) & r(X) & s(X) ).
```

or, with the same meaning as the CGE above, as

```
p(X,Y) :- (ground(X) -> q(X,Y) & r(X) & s(X)
                      ;  q(X,Y),  r(X),  s(X) ).
```

or, perhaps, and still within &-Prolog, as

```
p(X,Y) :- (ground(X) -> q(X,Y) & r(X) & s(X)
                      ;  q(X,Y), (ground(X) -> r(X) & s(X)
                                              ;  r(X),  s(X) )).
```

In the current RAP-WAM system, the task of parallelizing a given program (performing rewritings of it such as those above) is in general performed automatically by the RAP-WAM compiler, based either on local (clause-level) analysis, or on a global, abstract interpretation-based analysis [15] that often makes run-time independence checks unnecessary. The full power of &-Prolog is, however, also available to the user for manual parallelization if so desired. For the purposes of this paper it will be assumed that the program is already annotated with CGEs. Also, it will be assumed that such CGEs are not nested. This can be done without loss of generality by assuming that nested CGEs are "pulled out" as separate clauses. Furthermore, for simplicity, only unconditional CGEs (i.e. where *i_cond* is `true`) will be used. The method, however, is just as valid for any &-Prolog clause, as shown in section 4.1.

2.2 Some Terminology

- A Prolog builtin predicate which has side-effects is called a *side-effect builtin (seb).*
- Furthermore, these sebs are classified as *soft-sebs* and *hard-sebs.* Soft-sebs are those side-effects which do not affect the following computation (e.g *write*). Hard-sebs are those side-effects which alter the contents of the prolog database and hence *can* affect the following computation (e.g. *assert*).
- A clause that has at least one subgoal which is either a side-effect builtin or a side-effect procedure (*sep*) is called a *side-effect clause(sec).* Furthermore, it is classified as a *hard-sec* if it has at least one subgoal which is a hard-seb or a hard-sep. Otherwise, it is classified as a *soft-sec.*
- A procedure that contains at least one side-effect clause is called a *side-effect procedure(sep).* It is classified as a *hard-sep* if it contains at least one hard-sec. Otherwise, it is classified as a *soft-sep.*
- Procedures which are neither *sebs* or *seps* are classified as *pure* procedures.

2.3 Synchronizing Side-Effects with Semaphores

As mentioned before, one approach towards ensuring that the order of execution is preserved is to add some sort of synchronization code to the program. DeGroot's solution [5] is based on the use of "synchronization blocks." In DeGroot's description a synchronization block (*sb*) consists of two memory words: the first word (*sb.ecnt*) is used to maintain a count of goals involved in the synchronization, while the second word(*sb.signal*) is used for signaling the completion of a side-effect predicate. These synchronization blocks are presumably allocated from a special memory area devoted to this purpose. DeGroot also defines abstract operations which create and manage these blocks.

We will describe our method by making incremental modifications to DeGroot's.[2] The first step is to make use of normal Prolog objects to implement the synchronization blocks. Let us in principle define a "Synch-Block" as the Prolog structure `(Signal,Ecnt)`. This is the same as DeGroot's synchronization blocks except that synch-blocks are standard *Prolog structures* containing *logical variables.* This helps in two ways:

- This method has the advantage that a small set of new Prolog builtins is all that is required to implement all synchronization operations. These are described later in this section. Also, semaphores are allocated from (and integrated into) the normal Prolog storage model. Finally, no

[2]In attention to the reader, the explanation will be however self-contained so that no prior knowledge of DeGroot's method is required.

```
:- foo.
foo :- s1 & s2.
s1 :- a & s3.
s2 :- b & s4 & c.
a :- d & e.
s3 :- f & sse & g.
b :- h.
s4 :- j & hse.
c :- k & l.
```

Figure 1: Example program with side-effects

change in the syntax of the language is required: annotation of a clause with semaphores can be performed as a rewriting of the clause.

- In addition, correct backtracking behaviour is guaranteed because the Synch-Blocks are integrated within the standard backtracking machinery. No additional program annotation is needed for implementing backtracking in the presence of side-effects. This is explained in section 3.

Now, let us analyze how synch-blocks are actually used. As explained in DeGroot [5], the variable `Signal` takes on the values `yes` and `no` and the variable `Ecnt` takes on values from the set of natural numbers. Informally speaking, `Signal` indicates whether a side-effect builtin has finished executing (`yes` or `no`) and `Ecnt` has the count of the number of "pure" subgoals that are yet to finish executing before a side-effect builtin can start executing.

We now show how the synch-block structure can be simplified. As a first step, let us focus on the values taken by `Signal`. We replace `yes` with the number 1 and `no` with the number 0. Next, we define a semaphore `Sem` equivalent to the synch-block `(Signal,Ecnt)` as follows:

```
Sem = 2 * Ecnt + Signal
```

Observe that the values of `Signal` and `Ecnt` can be obtained from `Sem` as follows:

```
Signal = Sem mod 2; Ecnt = Sem div 2
```

So, essentially, the semaphore `Sem` carries the same information as the synch-block `(Signal,Ecnt)`, but is simpler in structure, being a single logical variable.

To illustrate how semaphores can be used to synchronize side-effects, consider the example program in Figure 1. For the sake of clarity, all the CGEs have their `i_cond`s equal to `true` and also, the predicates do not have any arguments. In this program, `sse` is a soft-seb and `hse` is a hard-seb. `s1, s2,`

Figure 2: Execution tree and arrangement of segments

---

(1) Subgoals in *pure1* and *pure2* can together be executed in parallel.
(2) Subgoal *sse* can be started only after the subgoals in *pure1* have been executed.
(3) Similarly, subgoal *hse* can be started only after *sse* and the subgoals in *pure1* and *pure2* have been executed.
(4) Finally, since *hse* is a hard side-effect builtin, the subgoals in *pure3* have to wait for the completion of *hse*.

Figure 3: Conditions for proper execution of program in Figure 1

`s3, s4` are side-effect procedures and `a, b, c, d, e, f, g, h, j, k, l` are 'pure' procedures.

This program has an execution tree as shown in Figure 2. It can be seen that the leaves of this tree can be divided into alternating segments: pure1(`d, e, f`), then a soft side-effect builtin(`sse`), then pure2(`g,h,j`), then a hard side-effect builtin(`hse`) and finally, pure3(`k,l`). Figure 3 lists the conditions to be satisfied for the proper execution of the program.

To satisfy the conditions in Figure 3 and still attain maximum parallelism in the execution of this program, three semaphores `Sem1, Sem2, Sem3` are added to it. (Informally, one can think of `Sem1` as the semaphore for the subgoals in *pure1* and `Sem2` as the semaphore for the subgoals in *pure2* and `Sem3` as the semaphore for the subgoals in *pure3*).

When the program is started, `Sem1` = 1. This is because *pure1* has no side-effect builtin to its "left" and so the subgoals in *pure1* can start executing immediately. But, since `sse` and `hse`, the side-effect builtins to the "left" of

pure2 and *pure3* respectively, are not yet finished, `Sem2` and `Sem3` start out with the value 2. This indicates that the side-effects `sse` and `hse` are not yet done.

Now, the values of the semaphores are changed as follows:
`Sem1 := Sem1 + 3*2` (for the 3 subgoals in pure1)
`Sem2 := Sem2 + 3*2` (for the 3 subgoals in pure2)
`Sem3 := Sem3 + 2*2` (for the 2 subgoals in pure2)

The subgoals in pure segments wait for the values of the corresponding semaphores to become *odd* and then they start executing. The side-effect builtins wait for the corresponding semaphores to be equal to 1 and then they start executing. For example, the subgoals in the segment pure1 (i.e. `d,e,f`) wait on `Sem1` being odd. Since this condition is true when the program is started, subgoals `d,e,f` immediately start executing in parallel. Note that none of the side-effect builtins (`sse` and `hse`) or the pure segments(*pure2, pure3*) can be executed at this time, because their waiting conditions are not satisfied. After each of the subgoals in *pure1* has been executed, `Sem1` is decremented by 2, so that when all the 3 subgoals are done with their execution, `Sem1` has the value 1. Now, the side-effect builtin `sse` which was waiting on (`Sem1` = 1), starts executing and when it is done, decrements `Sem2` by 2 and makes the value of `Sem2` odd (by incrementing `Sem2` by 1 if it is even and by doing nothing if it is already odd).This triggers the execution of the subgoals in the segment *pure2* which were waiting on `Sem2` being odd. The same pattern of execution is repeated. In summary, the execution order of the subgoals in the leaves of Figure 2 follows the conditions in Figure 3. The conditions on which the various segments wait are illustrated in Figure 4.

Observe that even though condition 1 in Figure 3 states that subgoals in segments *pure1* and *pure2* can be executed in parallel (since `sse` is a soft side-effect builtin), in the execution sequence that we have described so far, *pure1* is first executed in parallel and then, *pure2* is executed in parallel. So, we have not achieved maximum potential parallelism in the execution of the subgoals. To realize this goal, all we have to do is to make `Sem2` odd at the beginning. This triggers the parallel execution of the subgoals in *pure2* immediately.

Before we annotate this program with semaphores, we need to define certain operations on them. These operations are as follows:

`wait_odd(Sem)`	The current process waits on `Sem` being odd.
`wait_one(Sem)`	The current process waits on `Sem` = 1.
`inc2(Sem,N)`	Backtrackable, atomic increment: increments `Sem` by 2N.
`dec2(Sem)`	Backtrackable, atomic decrement: decrements `Sem` by 2.
`make_odd(Sem)`	Backtrackable, atomic operation: Increments `Sem` by 1 if it is even, else no-op.

All these operations have backtracking semantics, which are described in section 3. Figure 5 shows the program in Figure 1 annotated by semaphores.

Figure 4: Synchronization between segments

3 Backtracking in the Presence of Side-Effects

In this section, we show how, in the model presented herein, backtracking can still be supported while correctly sequencing the side-effects.

Consider the following program:

```
   :- s1.
s1 :- a, ( b & s2 & c ), d.
s2 :- e & hse & f.
```

Here, `a,b,c,d,e,f` are pure procedures and `hse` is a side-effect builtin. Thus, `s1, s2` are side-effect procedures. First, let's consider how backtracking would proceed in the normal case, i.e. if `hse` were not a side-effect. We assume in principle that the backtracking method presented in [9, 7] is used. This method dictates that if after the CGE `( b & s2 & c )` is entered one of `b, s2` or `c` fails, then the other two goals will be "killed" and execution will return to the next alternative of `a` ("inside" backtracking). If, however, the CGE is exited successfully and `d` fails, the rightmost of `b, s2, c` with alternatives will be found, redone, and execution will proceed in parallel again with the rest of the goals in the CGE to the right of the one being redone.

Since, on the other hand, `hse` is a hard side effect and, therefore, `s2` a side-effect procedure, the program would be annotated as follows:

```
:- Sem1 = 1, Sem2 = 2, s1(Sem1,Sem2).
```

```
:- Sem1 = 1, Sem3 = 2, foo(Sem1,Sem3).
foo(Sem1,Sem3):-Sem2 = 2,
                (s1(Sem1,Sem2) & s2(Sem2,Sem3)).
s1(Sem1,Sem2):- inc2(Sem1,1), make_odd(Sem2),
                ((wait_odd(Sem1),a,dec2(Sem1)) & s3(Sem1,Sem2)).
s2(Sem2,Sem3):- inc2(Sem2,1), inc2(Sem3,1),
                ((wait_odd(Sem2),b,dec2(Sem2)) & s4(Sem2,Sem3) &
                (wait_odd(Sem3),c,dec2(Sem3))).
a :- d & e.
s3(Sem1,Sem2):- inc2(Sem1,1), inc2(Sem2,1),
                ((wait_odd(Sem1),f,dec2(Sem1)) &
                (wait_one(Sem1),sse,dec2(Sem2),make_odd(Sem2)) &
                (wait_odd(Sem2),g,dec2(Sem2))).
b :- h.
s4(Sem2,Sem3):- inc2(Sem2,1),
                ((wait_odd(Sem2),j,dec2(Sem2)) &
                (wait_one(Sem2),hse,dec2(Sem3),make_odd(Sem3))).
c :- k & l.
```

Figure 5: Program annotated with semaphores

```
s1(Sem1,Sem2) :- a, inc2(Sem1,1), inc2(Sem2,1),
                 ((wait_odd(Sem1),b,dec2(Sem1)) &
                  s2(Sem1,Sem2) &
                  (wait_odd(Sem2),c,dec2(Sem2))
                 ),d.
s2(Sem1,Sem2) :- inc2(Sem1,1), inc2(Sem2,1),
                 ((wait_odd(Sem1),e,dec2(Sem1)) &
                  (wait_one(Sem1),hse,make_odd(Sem2),dec2(Sem2))
                  & (wait_odd(Sem2),f,dec2(Sem2))
                 ).
```

It is important to note that, as mentioned before, `dec2(Sem)`, `inc2(Sem, Int)` and `make_odd(Sem)` are "backtrackable" procedures, i.e., they undo the effects created by them on their parameters when they backtrack. The backtracking behavior of these procedures is described as follows:

`make_odd(Sem)`	does nothing if Sem is already even, else atomically decrements Sem by 1 if Sem is odd.
`dec2(Sem)`	atomically increments Sem by 2.
`inc2(Sem,N)`	atomically decrements Sem by 2N.

Note that the backtracking behaviors of these procedures are the reverse of what they do in the forward direction, as it should be. Also note that no

Figure 6: Implementation of Backtrackable Increment and Decrement

backtracking behavior needs to be described for the "waiting" procedures `wait_odd` and `wait_one`. One way of implementing this backtracking behaviour in a WAM-based, parallel abstract machine such as the RAP-WAM [7] is by having each worker record in its trail the actions performed by it on the semaphores, as shown in figure 6. This figure shows two snapshots of the trail of two workers, one just before the CGE in `s1` is entered, and the other one while the execution of `b` and `s2` is proceeding in parallel. Note that this implementation is similar to that of `setarg` in SICStus Prolog [1], but the *nature* of the action (i.e. increment, decrement, etc.) rather than the previous value is saved. This is recorded as a tag for the trail entry (I - increment, D - decrement, A - normal trail entry, etc.), the rest of the entry being the address of the semaphore. This is important since increments and decrements can occur in any order. A value – the amount of increment – is saved, however, for the `inc2` primitive. Also note that, depending on the optimizations implemented in the abstract machine, the variables that are used as semaphores should be marked as permanent (or even moved to the heap) to prevent possibly incorrect deallocation from last call- and other possible optimizations.

Now we describe how backtracking is done in two cases viz., (1) when `d` fails (case 1) and (2) when `c` fails (case 2).

- In **case 1**, assume that the subgoal `e` has unexplored alternatives, while `hse, f, c` do not. When `d` fails, alternatives for the subgoals in the CGE have to be explored and this is done with the subgoals being considered in the right-to-left order. Unwinding of the subgoal `c` leads to `Sem2` = 3, with `Sem1`'s value not being changed. Unwinding of the subgoal `s2` leads to unwinding of the subgoals `e, hse, f`. This leads to `Sem1` = 3 and `Sem2` = 6. Now, the subgoals `e, hse, f, c` are all restarted in parallel and it is clear that parallelism is obtained, maintain-

ing at the same time the sequential behavior necessary in the presence of side-effects. If the CGE succeeds, the subgoal `d` can be tried again.

- In **case 2**, assume that the subgoal `a` has unexplored alternatives. Since `c` has failed, the CGE cannot succeed, so backtracking has to be done for all the subgoals in this CGE, i.e. for `b, s2, c`. Note, however, that `b` and `s2` cannot be simply "killed" because in the sequential model `s2` would have completed and produced its side-effect before `c`'s failure. The idea again is to mimic the behaviour of the sequential model. A conservative modification of the "inside" backtracking actions which assures compatibility with the sequential model follows: if a goal fails, all goals to its right in the CGE are killed as before. Also, all goals to the left of the failing goal up to the previous side-effect goal are killed. The rest of the goals are allowed to continue to completion, after which they are all backtracked (including the side-effect goal, which may have other alternatives) by recursively applying the algorithm. It is clear from the backtracking behaviors of the synchronization procedures that after this is done, `Sem1` and `Sem2` are restored to their respective values of 1 and 2. Now `a` succeeds with an alternative binding for the variables and the CGE is reexecuted with the correct initial values for the semaphores.

Thus, backtracking in the presence of side-effects is relatively simple in this model. Nothing special is required because the semaphores are implemented as *logical variables*. Some optimizations can be made on this model. For example, in **case 1**, if the subgoal `c` has unexplored alternatives, no backtracking has to be done on `Sem2` and also, the re-execution of the subgoal `c` need not be accompanied by any waiting or decrementing operation on `Sem2`. But implementing this optimization will require primitives in addition to the ones described in Hermenegildo [7].

4 Optimizations and Other Issues

4.1 When the *i_cond* is non-empty

Our previous discussion was limited to CGEs with `i_cond = true`. In this section, we show how a *conditional* CGE is annotated with semaphores. Consider the following clause:

```
s1 :- (i_cond => a & se1 & b & se2 & c).
```

`a, b, c` are pure procedures and `se1` and `se2` are side effect builtins. If `se1` and `se2` were not side-effect builtins, this clause would have been annotated as:

```
( i_cond -> a & se1 & b & se2 & c
          ; a , se1 , b , se2 , c )
```

The presence of side effects changes the annotation as follows:

```
s1(Sem1,Sem2) :-
        ( i_cond ->
            Sem3 = 2, inc2(Sem1,1),
            inc2(Sem2,1), inc2(Sem3,1),
            ( (wait_odd(Sem1),a,dec2(Sem1)) &
              (wait_one(Sem1),se1,make_odd(Sem3),dec2(Sem3)) &
              (wait_odd(Sem3),b,dec2(Sem3)) &
              (wait_one(Sem3),se2,make_odd(Sem2),dec2(Sem2)) &
              (wait_odd(Sem2),c,dec2(Sem2)) )
         ; a, se1, b, se2, c, dec2(Sem2), make_odd(Sem2) ).
```

Let us call the code that is executed if `i_cond` succeeds as `parallel code` and the code that is executed if `i_cond` fails as `sequential code`. The following points are worth noting:

- The `parallel code` needs the use of a new semaphore `Sem3` (because of the two side-effects in the body), while the `sequential code` doesn't.
- While the `parallel code` has several synchronization primitives operating on the various semaphores, the `sequential code` operates only on `Sem2` and this is just to signal that the side-effect builtin corresponding to `Sem2` has been completed.

Several such optimizations can be used while annotating a given program with semaphores.

4.2 When Side-Effects Don't Need to be Synchronized

Another observation which can be made is that sometimes the user doesn't really care in which order side-effects are executed, i.e. the user doesn't need the parallel system to produce identical results (and in the same order) to those of sequential implementation. For this purposes we propose to include a whole new set of "parallel built-in side-effect predicates," for which no synchronization code is generated. For example, there would be a `p_write/1` version of the standard `write/1` predicate, and similarly for other side-effect built-ins. An example showing the use of the `p_format/3` predicate is shown below. This predicate locks access to the stream that it is writing to but is otherwise not synchronized with other builtins. Note that in the example, since each `do_`x parallel goal identifies its output in the file, we don't care in which order each of them writes into the file. The introduction of parallel side-effects such as `p_format/3` allows for more parallelism (and less overhead due to synchronization chores) that is attainable if the sequential Prolog semantics is enforced.

```
p(X,FileName)    :-
        ( ground([X,FileName]) => do_a(X,FileName) &
                                  do_b(X,FileName) &
                                  do_c(X,FileName) ).
```

```
do_a(X,FileName) :-
        do_a_lot(X,Result),
        p_format('Result of a = %i',[Result],FileName).
do_b(X,FileName) :-
        do_b_lot(X,Result),
        p_format('Result of b = %i',[Result],FileName).
do_c(X,FileName) :-
        do_c_lot(X,Result),
        p_format('Result of c = %i',[Result],FileName).
```

4.3 Multiple Synchronization Chains

Sometimes, even if the standard Prolog behaviour is to be preserved, it turns out that some aspects of the order in which side-effects are executed are not observable and therefore the synchronization requirements can be relaxed. Consider, for example, the case when parallel goals are writing to two different files: only the order of writing *within each file* needs to be preserved. However, it is still required that the "pure" procedures preceding the side effects be completed before the side effects are themselves executed. Consider the following example:

```
s1(X,Y) :-  ( ground([X,Y]) => a(X) & write(file1,X) &
                                b(Y) & write(file2,Y) ).
```

Here, `a(X), b(Y)` are pure procedures. Since there are two side effects in this CGE, we need three semaphores, `Sem1, Sem2, Sem3`. Normally `Sem2` has the initial value of 2. This forces `write(File2,Y)` to follow `write(File1,X)`. However, in this case, there is no need to sequentialize the two `write` statements since they write to *different* files. So, we can intialize `Sem2` to 0. This has the effect that it does not sequentialize the two `write` statements. This is a very useful optimization especially if the `write` statements take a long time to complete.

The following example further illustrates the point:

```
s1(X,Y) :- (ground(X) => foo(X) & a(X) & bar(Y) & b(X) ).
a(X) :- do_a_lot(X,Res1,Res2),
        write(Res1,file1), write(Res2,file1).
b(X) :- do_b_lot(X,Res1,Res2),
        write(Res1,file2), write(Res2,file2).
```

Other optimizations beyond those presented in sections 4.3 and 4.2 are described in [13].

5 Two Alternative Methods Using Semaphores

In this section, we describe two different variations on the same idea of implementing semaphores. The first method's objective is to avoid introducing semaphores as additional arguments for the side-effect predicates. The second method encodes the information in synch-blocks in a different way from what

was described before. These methods and the associated program transformations are described in more detail in [13].

5.1 Method 1: Reducing the Argument-Passing Overhead

In section 2.3, semaphores were introduced as additional arguments for the side-effect predicates. This results in

- `compile time overhead` – additional work done in adding the semaphores as arguments to the side-effect predicates.
- `runtime overhead` – more arguments for the side-effect predicates imply a longer runtime for the program, even though they contribute nothing to the solution of the underlying problem.

In this method, semaphores are not introduced as additional arguments for the predicates, but are stored in a *global list data structure*, called the `semaphore list`. This scheme can be easily implemented in a shared memory parallel processor.

A separate chunk of memory is earmarked for the `semaphore list`. A program which has side-effect predicates is started with the `semaphore list` having two semaphores (in that order), `Sem1` which has the value 1 and `Sem2`, which has the value 2. A side-effect predicate which is called in the body of a clause, has access to a pointer `Local Head`, which points to a semaphore in the `semaphore list`. The following "macros" are used in this method.

- `side_effect(Pred,P)` calls the side-effect procedure `Pred` after setting its `Local Head` to the pointer `P`.This points to the 'leftmost' semaphore for `Pred`.
- `pure(Pred,P)` – `P` points to a semaphore, say `Sem`. This procedure is defined as follows: `pure(Pred,P) :- wait_odd(Sem), Pred, dec2(Sem).`
- `add_semaphores(N)` – creates `N` semaphores and inserts them in the `semaphore list`. If `Sem1` is the semaphore pointed to by `Local Head` and the next semaphore in the `semaphore list` is `Sem2`, this procedure inserts `N` semaphores between `Sem1` and `Sem2`.

As an example of how this method is used, consider the following clause:

```
s1 :- a & s2 & b & s3 & c.
```

`s1, s2` and `s3` are side-effect procedures and `a, b` and `c` are 'pure' procedures. This clause is annotated as follows:

```
s1 :-  add_semaphores(1),
       (  pure(a,1) &
          side_effect(s2,1) &
          pure(b,2) &
          side_effect(s3,2) &
          pure(c,3)
       ).
```

Figure 7: How Synchronization Blocks help achieve synchronization

Figure 8: Construction of the semaphores in Method 2

This method is attractive, although the use of additional global (heap) storage should be traded off with the reduction in procedure calling time.

5.2 Method 2: An Alternative Encoding for Synch-Blocks

Consider the side-effect builtin in Figure 7. This has one synch block(SB1) to its 'left' and another synch block(SB2) to its 'right'. Now, take the 'ecnt' record from SB1 and the 'signal' record from SB2 and put them together and encode them in one semaphore, say `Sem2`. This is done for all the side-effect builtins. The 'leftmost' semaphore, of course, will have only the 'signal' part of a synch block and the 'rightmost' semaphore will have only the 'ecnt' part of a synch block. Figure 8 shows the constructions of the semaphores for the program segments in Figure 7.

How is the information in SB1_ecnt and SB2_signal encoded into just one record in Sem2? This is done as follows: Before the execution of the program, Sem2 is created (with value = 0) and then incremented by the number of subgoals in *pure1*. After each 'pure' subgoal in *pure1* has finished executing, it decrements Sem2 by 1. Meanwhile, 'side-effect builtin' waits on (Sem2 = 0) for its execution. This will start executing once all the subgoals in *pure1*

are done. Sem2 is decremented by 1 (thus making its value = -1), after the execution of the 'side-effect builtin' is completed. The subgoals in *pure2* which were waiting on (Sem2 = -1) now start executing in parallel. This method offers the advantage of conceptual (and implementation) simplicity, but requires more semaphores than that described in section 2.3. Also, in that method, it is possible to execute the subgoals in a pure segment immediately following a soft side effect builtin *in parallel*, even before that builtin has been executed. This cannot be done in method 2.

6 Conclusions

We have presented an efficient and complete method for implementing Independent/Restricted And-Parallelism in the presence of side effects. This method does not suffer from the drawbacks of previously proposed solutions, i.e (1) it supports backtracking, (2) it uses a simple data structure for semaphores and a compact set of primitives on the semaphores, (3) it allows more parallelism, (4) it deals with the case of having a mixture of parallel and sequential goals, and (5) since it implements semaphores as *logical variables*, it can be easily added to a parallel abstract machine-based system through only minor modifications. We have also presented two additional methods for implementing And-Parallelism in the presence of side effects. These methods, an additional and rather interesting method (based on keeping track of the execution graph), as well as the algorithms for detecting side-effect procedures and annotating them with semaphores are described in detail in [13].

7 Acknowledgements

The authors would like to thank the other members of the PAL group, Kevin Greene and Roger Nasr, for their comments on earlier drafts of this paper.

References

[1] M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the Wam. In *4th Int. Conf. on Logic Prog.*, pages 40–58. MIT Press, May 1987.

[2] J.-H. Chang, A. M. Despain, and D. Degroot. And-Parallelism of Logic Programs Based on Static Data Dependency Analysis. In *Compcon Spring '85*, pages 218–225, February 1985.

[3] J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

[4] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.

[5] D. DeGroot. Restricted AND-Parallelism and Side-Effects. In *International Symposium on Logic Programming*, pages 80–89. San Francisco, IEEE Computer Society, August 1987.

[6] B. Hausman, A. Ciepielewski, and A. Calderwood. Cut and Side-Effects

in Or-Parallel Prolog. In *International Conference on Fifth Generation Computer Systems*. Tokyo, November 1988.

[7] M. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*. PhD thesis, U. of Texas at Austin, August 1986.

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

[9] M. Hermenegildo and R. I. Nasr. Efficient Management of Backtracking in AND-parallelism. In *Third International Conference on Logic Programming*, number 225 in LNCS, pages 40–55. Imperial College, Springer-Verlag, July 1986.

[10] M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. Technical Report ACA-ST-032-89, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, January 1989.

[11] L. V. Kalé. Completeness and Full Parallelism of Parallel Logic Programming Schemes. In *Fourth IEEE Symposium on Logic Programming*, pages 125–133. IEEE, 1987.

[12] Y.-J. Lin, V. Kumar, and C. Leung. An Intelligent Backtracking Algorithm for Parallel Execution of Logic Programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 55–69. Imperial College, Springer-Verlag, July 1986.

[13] K. Muthukumar and M. Hermenegildo. Efficient Methods for Supporting Side Effects in Independent And-parallelism and Their Backtracking Semantics. Technical Report ACA-ST-031-89, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, January 1989.

[14] D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

[15] R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.

[16] H. Westphal and P. Robert. The PEPSys Model: Combining Backtracking, AND- and OR- Parallelism. In *Symp. of Logic Prog.*, pages 436–448, August 1987.

[17] W. Winsborough and A. Waern. Transparent And-Parallelism in the Presence of Shared Free variables. In *Fifth International Conference and Symposium on Logic Programming*, pages 749–764, Seattle,Washington, 1988.