facultad de informática

universidad politécnica de madrid

Conditional Parallelization of Non-Strict Independence: Procedures and Assessment

Daniel Cabeza Gras
Manuel Hermenegildo

Conditional Parallelization of Non-Strict Independence: Procedures and Assessment

---

Authors

Daniel Cabeza Gras
dcabeza@dia.fi.upm.es
Manuel Hermenegildo
herme@fi.upm.es

Departamento de Inteligencia Artificial
Facultad de Informática
Universidad Politécnica de Madrid (UPM)
28660-Boadilla del Monte, Madrid - SPAIN

Keywords

Parallel Execution of Logic Programs, Compilation Techniques, Generation of Annotations for Parallelism, Abstract Interpretation, Non-strict Independent And-Parallelism.

Abstract

This paper presents a conditional parallelization process for and-parallelism based on the notion of non-strict independence, a more relaxed notion than the traditional of strict independence. By using this notion, a parallelism annotator can extract more parallelism from programs. On the other hand, the intrinsic complexity of non-strict independence poses new challenges to this task. We report here on the implementation we have accomplished of an annotator for non-strict independence, capable of producing both static and dynamic execution graphs. This implementation, along with the also implemented independence checker and their integration in our system, have resulted what is, to the best of our knowledge, the first parallelizing compiler based on non-strict independence which produces dynamic execution graphs. The paper also presents a preliminary assessment of the implemented tools, comparing them with the existing ones for strict independence, which shows encouraging results.

1 Introduction

Several types of parallel logic programming systems and models exploit and-parallelism among non-deterministic goals [5, 4]. This parallelism relies on notions of independence among those goals in order to ensure certain efficiency properties. Two basic notions of independence are strict and non-strict independence [12, 8, 9]. Non-strict independence is a more relaxed notion than the traditional notion of strict independence which still ensures the relevant efficiency properties and can allow considerable more parallelism. However, all compilation technology developed to date had been based on strict independence, because of the intrinsic complexity of exploiting non-strict independence. In this paper we present, in the context of ESPRIT project #6707 *ParForce*, the implementation of an annotator based on the notion of non-strict independence, capable of producing both static and dynamic execution graphs. The techniques presented here are complemented with the ones shown in [2] (ParForce attachment D.WP1.2.1.M2.2). We show also a preliminary assessment of the techniques proposed, fully implemented in a parallelizing compiler, which complements likewise the abovementioned attachment and the ParForce attachment D.WP2.3.1.M2.3.

2 The Parallelization Process of Non-Strict Independence

In order to exploit parallelism based on the notion of non-strict independence, and contrary to strict independence, a preceding global analysis of the program is required. Then follows an annotation process, which applies, for the Sharing+Freeness abstract domain (or an equivalent one), the conditions presented in [2]. This process, presented in the following section, is different from that used for strict independence [1]. Finally, and unlike with strict independence, a process of renaming and substitution of shared variables is needed, explained in section 2.2.

2.1 Non-Strict Independence Annotation

The annotation process consists of, given a notion of independence, identifying opportunities for parallel execution in a program, and rewriting the program including parallel expressions. This process must be of a heuristic nature for several reasons. First, because of the fork-join nature of the *Ciao*–Prolog parallel operator `&/2` that we will use, which prevents in same cases the extraction of all the potential parallelism present in a clause (a set of more flexible parallel operators have been designed that overcome this limitation, see [3]). Second, because if and when run-time tests are included, their execution consumes time, and thus a compromise between amount of parallelism and number of tests must be found.

The annotation process for non-strict independence, as said before, must be accomplished after a global analysis of the program. This is so because, contrary to strict independence, non-strict independence is not an “a priori” condition, i.e. it cannot be

tested at run-time ahead of the execution of the goals, unless certain properties regarding the run-time instantiations of program variables by the goals are known. This fact conditioned the design of the algorithm for the annotation of non-strict independence, making it necessarily different from those used for strict independence, to suit the special characteristics of non-strict independence.

A restriction when annotating non-strict independence, not found with strict independence, is due to its asymmetrical nature. Strict independence annotators where free to rearrange the goals found independent even though they were not run in parallel, the symmetry of strict independence guaranteeing the safeness of this (in fact, the UDG and CDG algorithms do this reordering). This is not true anymore for non-strict independence, and thus the new annotator had to be designed in such a way so as to never reverse the execution order of two literals in a clause.

One decision in the design was to give priority to unconditional parallelism: if we know that two literals are independent without the need of run-time tests, we always should execute them in parallel, even if this prevents the exploitation of further conditional (that is, uncertain) parallelism. The MEL algorithm for strict independence, for example, fails to do this.

The non a priori nature of non-strict independence imposed another restriction: that the conditional parallelism can involve only pairs of goals. This is required because the run-time tests computed are based on properties about run-time instantiations of program variables in certain points of the program, and thus can only be safely placed in those points. For example, if we wanted to run in parallel three literals, the independence test for the last two is only applicable just before the second one, but we had to include it before the first one. Of course a reanalysis of the program should give the necessary information to compute the test in this case, but this seems impractical, unless perhaps if the analysis can be incremental [7].

But this last restriction has indeed practical repercussions: it keeps the conditional parallel expressions simple, and ensures that, if there exists parallelism in two contiguous atoms, something will run in parallel (this is not true with the CDG or MEL algorithms). We think that it is not worth making complex conditional expressions, which are costly to generate and costly to execute, when we have little information about the runtime values of the variables. Rather, it is preferable to possibly exploit less parallelism in a clause by executing less tests than to try to execute the most literals in parallel in the best case at the expense of having to check complex tests and risking losing all the parallelism in the not-so-good cases.

The next two sections explain the two steps that comprise the annotator. The first step identifies unconditional parallelism, that is, parallelism that does not need run-time tests. The second step, executed only when run-time tests are allowed, adds conditional parallelism to the parallel expression computed by the first step. This guarantees the objective of giving priority to unconditional parallelism over conditional parallelism.

2.1.1 *Unconditional Annotation: the URLP Algorithm*

The algorithm used in this first step is called URLP (Unconditional Recursive Linear Parallelizer), and gives for strict independence an amount of parallelism similar to UDG. But URLP does not use the dependency graph approach of UDG, since these graphs lose the information about the order of independent literals in the clause, thus allowing their reordering. Instead, the algorithm starts with the sequence of literals of the body of the clause, applying from left to right the following rewriting rules, until no change can be done:

Rule	Pattern	Condition	New Pattern
1	...A, B...	indep(A,B)	...A & B...
2	...PA, B...	(1)	...IA & (DA, B)...
3	...PA, PB...	(2)	...(PA, DB) & IB...

Where A and B represent literals and PA, PB, DA, DB, IA and IB represent literals or parallel expressions.

(1) IA (DA) is the parallel expression formed with the elements of PA from which B is independent (dependent). After the rule is applied, "DA, B" is parallelized recursively. The rule is only applied if IA is not empty.

(2) IB (DB) is the parallel expression formed with the elements of PB which are independent (dependent) from PA. The rule is only applied if IB is not empty.

The first rule is self explanatory: two independent contiguous literals can be executed in parallel. The second one states that if we have a parallel expression followed by a literal, then this literal can be executed in parallel with the elements of the parallel expression from which it is independent, but it must wait until the end of the goals on which it depends. The third rule is opposite to the second one: if we have a goal (either a literal or a parallel expression) followed by a parallel expression, then the elements of the expression which are independent from the goal can be executed in parallel with it, and the rest must wait until the goal ends.

Since at each step a simple rule is applied, it can be easily shown that the parallelization is correct. It makes no sense in general to discuss whether it is optimal, since for this we would need the execution times of the goals. As said before, the algorithm gives results similar to the UDG annotator.

It is important to emphasize an advantage of this algorithm over UDG. In UDG, to compute the dependency graph, every possible dependency between pairs of literals has to be computed. In URLP, however, some checks can be saved, since for example if we have the body clause "a, b, c" and we have found that b depends on a and that c depends on b, we do not need to check the possible dependency between a and c.

As an example, let's consider the parallelization of a body clause, which we will schematically denote as "a, b, c, d, e, f, g", where the dependencies that the conditions

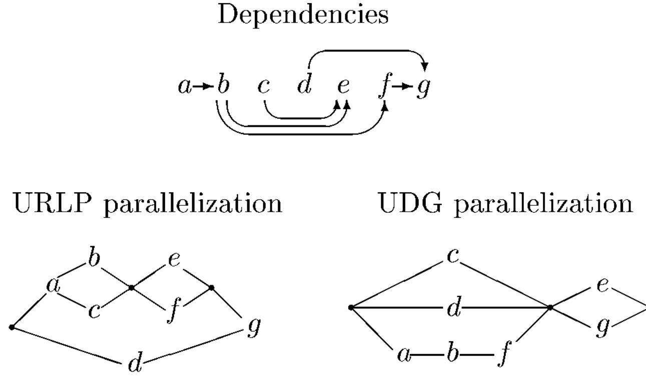

Figure 1: Example of parallelization with URLP and UDG.

for non-strict independence give are dep(a, b), dep(b, e), dep(b, f), dep(c, e), dep(d, g) and dep(f, g). The steps followed by the algorithm are shown below:

Step	Expression	Apply Rule #
0	a, b, c, d, e, f, g	1 in b, c
1	$a, b\&c, d, e, f, g$	2 in $b\&c, d$
2	$a, b\&c\&d, e, f, g$	2 in $b\&c\&d, e$
3	$a, d\&(b\&c, e), f, g$	2 in $d\&(b\&c, e), f$
4	$a, d\&(b\&c, e, f), g$	1 in e, f
5	$a, d\&(b\&c, e\&f), g$	3 in $a, d\&(b\&c, e\&f)$
6	$(a, b\&c, e\&f)\&d, g$	

The parallelization provided by UDG is $c\&d\&(a, b, f), e\&g$, reversing the relative order of execution of the goals e and f, which is what must be avoided for Non-Strict independence. Figure 1 shows the results.

2.1.2 Conditional Annotation: the CRLP algorithm

Once the URLP algorithm has computed an (unconditional) parallel expression, one more step is needed to be able to exploit further parallelism through the use of run-time tests. The two steps comprise what we call the CRLP algorithm (Conditional Recursive Linear Parallelizer).

The algorithm examines the input parallel expression to find sequences of literals not yet parallelized, and transforms them inserting the appropriate tests. Let *Exp* be one of these sequences, then **par(***Exp***)** is the transformed expression, defined as:

$$\mathbf{par}(\texttt{(p,q,...)}) = \begin{cases} \text{p, } \mathbf{par}(\texttt{(q,...)}) & \text{if } \mathbf{nsi}(\text{p,q}) = \texttt{false} \\ \text{p \& q, } \mathbf{par}(\texttt{...}) & \text{if } \mathbf{nsi}(\text{p,q}) = \texttt{true} \\ (\ \mathbf{nsi}(\text{p,q}) \rightarrow \ \text{p \& q, } \mathbf{par}(\texttt{...}) & \\ \qquad\qquad ;\ \text{p, } \mathbf{par}(\texttt{(q,...)})\ ) & \text{otherwise} \end{cases}$$

where **nsi(p,q)** is the test that ensures non-strict independence between literals `p` and `q`

[2], but having into account also the tests already verified by precedent expressions of the branch – this is why **nsi(p,q)** can be just `true`.

It can be easily shown that this simple algorithm meets the restriction that conditional parallelism involves only pairs of goals, and also that it ensures that if there exists parallelism in two contiguous atoms, something will run in parallel.

2.2 Renaming and Substituting Shared Variables

When using non-strict independence, and in order to prevent partial answers of a branch that will ultimately fail from pruning the search space of other goals, parallel goals are in principle run in independent environments (see [10]). The standard solution for this problem is a run-time transformation of the goals to be executed in parallel. This transformation involves eliminating any shared variable among parallel goals by renaming or substituting its occurrences so that no two occurrences in different goals remain the same, and adding some unification goals after the parallel conjunction to reestablish the lost links. This operation can be encoded at compile-time by performing `copy_term`'s of every goal and unifying the original goals and the copied versions after the parallel conjunction. We will now propose more efficient methods which are based on the knowledge gathered during the annotation process. Note that a mere renaming of variables at compile-time is not sufficient in general: we can have terms with shared variables inside. Thus, we use the following predicate:

$\texttt{subst_vars}([\texttt{X}_1,\ldots,\texttt{X}_n],[\texttt{X}'_1,\ldots,\texttt{X}'_n],\texttt{Z},\texttt{Z}')$:-
$\texttt{Z}'$ is a term equal to $\texttt{Z}$ but with variables $\texttt{X}'_1,\ldots,\texttt{X}'_n$ in place of variables $\texttt{X}_1,\ldots,\texttt{X}_n$, respectively.

It can be easily shown why this predicate is more efficient than the `copy_term` predicate: the latter copies all the term structure except perhaps ground substructures, whereas the former can also avoid copying non-ground substructures which do not contain variables to be renamed. Furthermore, using `copy_term`, when re-unifying the copies one has to unify the entire terms, whereas using `subst_vars` only the renamed free variables must be unified.

We are interested in the potential run-time shared variables, but with the conditions and/or the tests we ensure that these are the free variables (those of $\widehat{\beta}_{\mathrm{FR}}$) that appear in the sharing sets of SH (extending this concept to an arbitrary number of goals) — see [2] for an explanation of this concepts. Thus, the transformation procedure proceeds as follows:

- Group in sets the free variables that appear in the sharing sets of SH, so that those that appear in the same sharing set are grouped together, and the rest form sets with a unique element. This is so because if two free variables appear

in the same sharing set, they are possibly aliased at run-time, so they need to be processed together.

- For each of those sets of free shared variables V:
  - compute $\mathrm{R}(V) = \{w \mid \exists\, L \in \mathrm{SH}\ \exists\, v \in V\ \ v \in L \wedge w \in L \wedge w \notin V\}$, i.e. the set of the variables that appear in the sharing sets of SH with variables from V, excluding those of V. Thus they possibly contain at run-time variables from V.
  - Then, for each goal g, the necessary renamings or substitutions regarding V are computed. Let $\mathcal{V} = \mathrm{var(g)} \cap V$ and $\mathcal{R} = \mathrm{var(g)} \cap \mathrm{R}(V)$. We will represent a renaming of a variable v as "ren(v)" and a substitution of a variable v inside w as "sv(v, w)". There are three cases:
    * $\mathcal{V} = \emptyset, \mathcal{R} = \emptyset \rightarrow$ none.
    * $\mathcal{V} = \emptyset, \mathcal{R} \neq \emptyset \rightarrow \mathrm{sv}(v, w)$ for each $w \in \mathcal{R}$, where $v \in V$.
    * $\mathcal{V} \neq \emptyset \rightarrow \mathrm{ren}(v)$, $\mathrm{sv}(v, w)$ for each $w \in (\mathcal{R} \cup \mathcal{V} - \{v\})$, where $v \in \mathcal{V}$.
  - Since for each V we need to transform all the goals minus one, the goal with the most expensive transformation is not considered. Substitutions are more expensive than renamings; substitutions in ordinary variables are more expensive than substitutions in free variables (which are in fact conditional unifications).
- Once the transformations for all the sets of variables are computed, then for each goal the substitutions in the same variable are joined in a `subst_vars` predicate. Unification ("back-binding") goals must be included after the parallel conjunction for all the free variables renamed or substituted. Note that one side of these unifications is always a free variable, since the conditions ensure that the first goal do not instantiate shared variables.

As an example, consider the parallel expression `p(T,V,W) & q(U,V,W,X,Y) & r(W,Z)`, with the abstract call substitution $\widehat{\beta} = ([[\mathtt{T}]\,[\mathtt{UV}]\,[\mathtt{UVY}]\,[\mathtt{VWX}]\,[\mathtt{X}]\,[\mathtt{XY}]\,[\mathtt{Z}]], [\mathtt{TUWY}])$. The shared sharing sets are $\mathrm{SH} = [[\mathtt{UV}]\,[\mathtt{UVY}]\,[\mathtt{VWX}]]$. We have two sets of free variables from SH: $\{\mathtt{U},\mathtt{Y}\}$ and $\{\mathtt{W}\}$, with $\mathrm{R}(\{\mathtt{U},\mathtt{Y}\}) = \{\mathtt{V}\}$ and $\mathrm{R}(\{\mathtt{W}\}) = \{\mathtt{V},\mathtt{X}\}$. The following table shows, for each of these sets, and for each goal, the values of $\mathcal{V}$ and $\mathcal{R}$ and the transformation needed.

	$V = \{\mathtt{U},\mathtt{Y}\}$, $\mathrm{R}(V) = \{\mathtt{V}\}$			$V = \{\mathtt{W}\}$, $\mathrm{R}(V) = \{\mathtt{V},\mathtt{X}\}$		
	$\mathcal{V}$	$\mathcal{R}$	*transformation*	$\mathcal{V}$	$\mathcal{R}$	*transformation*
`p(T,V,W)`	$\emptyset$	{V}	sv(U, V)	{W}	{V}	ren(W), sv(W, V)
`q(U,V,W,X,Y)`	{U, Y}	{V}	ren(U), sv(U, Y), sv(U, V)	{W}	{V, X}	ren(W), sv(W, V), sv(W, X)
`r(W,Z)`	$\emptyset$	$\emptyset$	$\emptyset$	{W}	$\emptyset$	ren(W)

In both columns we discard the transformation for the goal `q/5`. The two substitutions for the goal `p/3` are on the same variable, so they must be joined. Therefore, the parallel expression is transformed into:

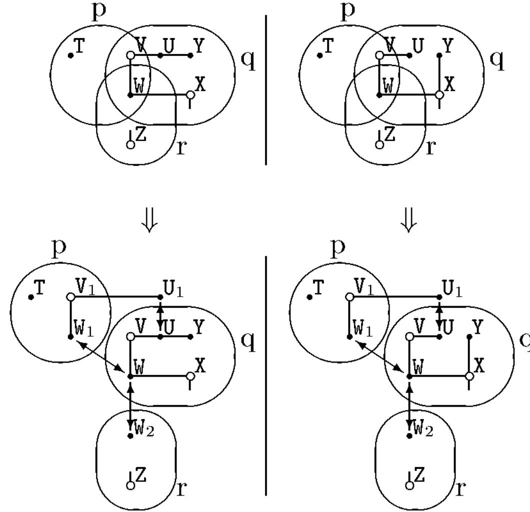

Figure 2: Representation of the effect of variable substitution in a parallel expression.

```
subst_vars([U,W],[U1,W1],V,V1),
p(T,V1,W1) & q(U,V,W,X,Y) & r(W2,Z),
U=U1, W=W1, W=W2
```

Figure 2, which uses the pictorial representation of abstract substitutions introduced in [2] (ParForce attachment D.WP1.2.1.M2.2), illustrates in pictures the transformation done, the bidirectional arrows showing the bindings performed by the back-binding goals. There are two situations depending on the covering of the free variables by the sharing sets.

3 Experimental results

We have integrated the URLP and CRLP parallelization algorithms in the *Ciao*–Prolog system [6] parallelizing compiler. Compiler switches determine whether or not code will be parallelized and, if so, through which type of analysis and annotator. In this section we present the results of the comparison between the new analyzers for non-strict independence (URLP and CRLP) and the existing ones for strict independence (MEL, CDG and UDG) [1]. In the experiments, we used the Sharing+Freeness global analysis [11], which is suited for non-strict independence parallelization. We used a relatively wide range of programs as benchmarks. For a detailed description of them see [1] (or ParForce attachment D.WP2.3.1.M2.3) and [2] (ParForce attachment D.WP2.3.1.M2.2).

Benchmark	MEL	CDG	UDG	URLP	CRLP
aiakl	242	238	238	237	239
ann	8915	8918	8920	8966	8941
array2list	479	475	476	477	478
bid	309	308	307	308	308
boyer	4323	4282	4269	4266	4272
browse	134	134	132	132	132
deriv	82	79	79	82	82
fib	63	58	58	16	16
flatten	60	59	58	57	60
grammar	123	121	122	120	119
hanoiapp	41	41	42	41	42
mmatrix	39	38	38	22	20
occur	47	48	48	45	48
progeom	169	168	169	168	169
qplan	1451	1452	1455	1446	1521
qsortapp	59	59	57	58	57
query	58	55	57	59	59
rdtok	2080	2104	2173	2100	2107
read	1974	2016	2016	2018	2006
serialize	661	657	662	660	660
sparse	104	103	107	105	105
tak	22	24	23	26	23
tictactoe	1240	1226	1235	1237	1249
warplan	8810	8831	8823	8897	8845
zebra	207	205	205	206	206

Table 1: Annotation Efficiency

3.1 Annotation Efficiency

Table 1 presents the results in terms of annotation times in milliseconds (SparcStation 10, four processors at 55MHz). It shows for each annotator the average time out of ten executions.

The table shows that the performance of the annotators, in terms of the time taken in annotating the programs, is very similar, without notable differences. This performance seems also reasonable.

Bench	Annotators	PAR	NSI	UNC	CON	LIT
boyer	UDG/URLP	0	0	0	0	0
	MEL/CDG/CRLP	2	0	0	5	4
browse	UDG/URLP	0	0	0	0	0
	MEL/CDG/CRLP	4	0	0	4	8
serialize	UDG/URLP	0	0	0	0	0
	MEL/CDG/CRLP	1	0	0	1	2

Table 2: Equal code for unconditional or conditional annotators

3.2 Annotation Results

From the benchmarks given in the previous table, the five annotators gave the same results for bid, deriv, fib, grammar, mmatrix, occur, progeom, qsortapp, query, rdtok, read, tak, tictactoe and zebra. This can be explained by observing that in general these benchmarks are either very simple or do contain little parallelism. We now show, for the rest of the benchmarks, the number of parallelized clauses extracted by each annotator (PAR), which of them exploit non-strict independence (NSI), which of them have unconditional parallelism (UNC), the total number of conditions for parallelization (CON), and the total number of literals that appear in parallel expressions (LIT).

Table 2 shows the results of the benchmarks whose parallelized code was different depending on whether the annotator was unconditional (UDG and URLP) or conditional (MEL, CDG and CRLP). This indicates that we can only exploit strict independence in those benchmarks, since using a non-strict independence parallelizer does not increase the amount of parallelism. Table 3 shows the results of the benchmarks whose parallelized code was different depending on whether the annotator was for strict independence (MEL, CDG and UDG) or non-strict independence (URLP and CRLP). In this case the benchmarks have non-strict independence, and all the extracted parallelism is unconditional. Finally, table 4 shows the results for the rest of the benchmarks. Not surprisingly, they are all complex programs with non-trivial parallelism, except hanoiapp, in which there is a slight loss of parallelism by using MEL.

3.3 Discussion

The information in the tables shows that the URLP and CRLP annotators are able to extract non-strict independence from several benchmarks, and furthermore that when the benchmarks have only strict independence they are able to extract roughly the same parallelism that their counterparts. More concretely, in the cases where all the parallelism is strictly independent, URLP behaves as UDG, and CRLP behaves in between MEL and CDG: it produces conditional expressions whose complexity is intermediate between those of the other two, but which possibly do not exploit all

Bench	Annotators	PAR	NSI	UNC	CON	LIT
aiakl	MEL/CDG/UDG	1	0	1	0	4
	URLP/CRLP	2	1	2	0	6
array2list	MEL/CDG/UDG	0	0	0	0	0
	URLP/CRLP	2	2	2	0	8
flatten	MEL/CDG/UDG	0	0	0	0	0
	URLP/CRLP	1	1	1	0	2
sparse	MEL/CDG/UDG	0	0	0	0	0
	URLP/CRLP	1	1	1	0	2

Table 3: Equal code for strict or non-strict annotators

Bench	Annotator(s)	PAR	NSI	UNC	CON	LIT
ann	UDG/URLP	0	0	0	0	0
	MEL	11	0	0	26	24
	CDG	11	0	0	35	29
	CRLP	11	0	0	31	24
hanoiapp	MEL	1	0	1	0	2
	the rest	1	0	1	0	3
qplan	UDG	15	0	15	0	38
	MEL	17	0	15	3	37
	CDG	17	0	15	3	42
	URLP	17	3	17	0	45
	CRLP	19	3	17	3	49
warplan	UDG/URLP	1	0	1	0	2
	MEL	7	0	1	16	22
	CDG	7	0	0	29	22
	CRLP	7	0	1	27	22

Table 4: Parallelization of the rest of the benchmarks

the parallelism in the clause. Note that, as discussed earlier, the algorithms must find a compromise between amount of parallelism and number of tests of the parallel expressions. In all, we think that the obtained results are quite encouraging.

4 Conclusions

We have presented parallelization techniques for exploiting conditional non-strict independent and-parallelism used in our system. They are the basis of the extension we have accomplished of the &-Prolog/*Ciao*–Prolog parallelizing compiler and system to produce and execute dynamic execution graphs. This allows conditional parallelism based on run–time checks, thus increasing the number of programs for which automatic exploitation of and-parallelism is feasible (and profitable). We have also presented an evaluation of these new tools for conditional non-strict parallelization. The results show encouraging performance. The new, conditional parallelizer completely subsumes the previous, strict independence based parallelizers, giving comparable results when the program exhibits strict independence and (obviously) improved results when the program exhibits non-strict independence. As future work, we plan to make use of more powerful analysis besides the Sharing+Freeness currently used, to further improve the amount of parallelism that can be exploited automatically. We believe that our current approach is valid for these more sophisticated types of analyses, either directly or with small modifications.

References

1. F. Bueno, M. García de la Banda, and M. Hermenegildo. A Comparative Study of Methods for Automatic Compile-time Parallelization of Logic Programs. In *Parallel Symbolic Computation*, pages 63–73. World Scientific Publishing Company, September 1994.

2. D. Cabeza and M. Hermenegildo. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. In Springer-Verlag, editor, *1994 International Static Analysis Symposium*, number 864 in LNCS, pages 297–313, Namur, Belgium, September 1994.

3. D. Cabeza and M. Hermenegildo. Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the 1995 COMPULOG-NET Workshop Parallelism and Implementation Technologies*, Utrecht, NL, September 1995. U. Utrecht / T.U. Madrid.

4. J. Chassin and P. Codognet. Parallel Logic Programming Systems. *Computing Surveys*, 26(3):295–336, September 1994.

5. 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.

6. M. Hermenegildo, F. Bueno, M. García de la Banda, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Proccedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming*, Portland, Oregon, USA, December 1995.

7. M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey. Incremental Analysis of Logic Programs. In *International Conference on Logic Programming.* MIT Press, June 1995.

8. M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.

9. M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.

10. M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.

11. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

12. 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.