PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT. © ICOT, 1992

Estimating the Inherent Parallelism in Prolog Programs

David C. Sehr [*] Laxmikant V. Kalé [†]
University of Illinois at Urbana-Champaign

Abstract

In this paper we describe a system for compile time instrumentation of Prolog programs to estimate the amount of inherent parallelism. Using this information we can determine the maximum speedup obtainable through OR- and AND/OR-parallel execution. We present the results of instrumenting a number of common benchmark programs, and draw some conclusions from their execution.

1 Introduction

In this paper we describe a method for timing Prolog programs by instrumenting the source code. The resulting program is run sequentially to estimate the sequential and best possible OR parallel execution times. This method is then extended to give the best possible AND/OR parallel execution time. Our instrumentation does not drastically reduce efficiency, and we present the results of a number of programs.

Our AND parallelism estimation method is based upon the work of by Kumar [1988] in estimating the inherent parallelism in Fortran programs. His method augments the source program with a timestamp for each data item *d*, which is updated each time *d* is written. In order to honor dependences, each computation that reads *d* can begin no earlier than the time recorded in *d*'s timestamp. The largest timestamp computed by such an augmented program is the *optimal parallel time* for the original program. This time can be used to evaluate how well a given implementation exploits parallelism.

This paper comprises six sections. The remainder of the first presents some terminology. The second describes measuring the amount of OR parallelism in a Prolog program. The third section extends this method to include AND parallelism. The fourth presents the timing methods used for several builtin predicates. The fifth section gives the results of our technique on the UCB Benchmarks. The last section presents some conclusions and suggests some future work.

1.1 Terminology

A prolog program consists of a *top-level query* and a set of *clauses*. The top-level query is a sequence of *literals*; we shall also use the term *query* to refer to any arbitrary sequence of literals. A literal is an *atom* or a *compound term* consisting of a *predicate name* and a list of *subterms* or *arguments*. Each subterm is an atom or a compound term. The number of subterms of a compound term is its *arity*. A clause has a *head literal* and zero or more *body literals*. A clause with no body literals is a *fact*; others are *rules*. Clauses are grouped into *procedures* by the predicate name and arity of their head literals. The rest of this paper assumes some working knowledge of Prolog's execution strategy.

For our timings we model a program's execution as traversal of its *OR tree* (*SLD tree*). Each node in an OR tree is labeled by a *query*. The first literal of the query at node N is the *literal at* N. The label of the root is the top-level query[1]. Each child N of a node M is produced by unifying a clause C's head with the literal L at M. N's query is formed by replacing L in M's query by the body of C. The left-to-right order of such children is the order of the clauses in the source program. A leaf node with an empty query is a *success*. Sequential Prolog systems traverse this tree depth-first and left to right.

[*]Center for Supercomputing Research and Development, 305 Talbot Laboratory, 104 S. Wright St., Urbana, IL 61801, USA. (sehr@csrd.uiuc.edu) This work was supported by Air Force Office of Scientific Research grant AFOSR 90-0044 and a grant from the IBM Corporation to CSRD.

[†]Department of Computer Science, Digital Computer Laboratory, 1304 W. Springfield Ave., Urbana, IL 61801, USA. (kale@cs.uiuc.edu) This work was supported in part by NSF grant NSF-CCR-89-02496.

[1]Which may have appeared in the source program, or may have been typed by the user at the read-evaluate-print prompt.

2 Sequential and OR time

The most efficient OR parallel implementations of Prolog to date [Warren 1987, Ali 1990] have been based upon the Warren Abstract Machine (WAM) [Warren 1983]. Because of this, we compute critical path timings in number of WAM instructions executed. The number of instructions is an approximation to execution time, since each type of WAM instruction takes a slightly different time. Variations in execution time come mainly from two sources: argument unification and backward execution. The former comes from the `get_value` and `unify_value` instructions, whose costs depend on the size of the terms they unify, which can be substantial. We address this by making the cost of these instructions the number of unification steps they perform. Backward execution comes from instruction failure and may perform significant bookkeeping changes, especially for deep backtracking. Different WAM implementations, particularly parallel ones, have differing costs for backward execution. In the measurements presented here we have assumed zero backward execution cost, but other cost assumptions can be used.

The execution time of a program has two components. The literal L at a node N in the OR tree is a call to a procedure p. Calling p consists of setting up L's calling arguments by a sequence of put instructions and performing the call by a `call` or `execute` instruction. The execution time of this sequence is a statically computable time $t_p(L)$ for L, which we approximate by the number of put instructions plus one.

Executing a called procedure consists of trying clauses in succession. If C is being tried for the call, the call arguments are *unified* with the arguments of C's head literal H. This is done by `get` and `unify` instructions and takes a time $t_u(H)$. In general the execution time of these instructions cannot be estimated at compile time, so this head unification is performed by calls to run-time routines for the corresponding WAM instructions. $t_u(H)$ is the sum of the times computed by these routines.

To represent execution times the OR tree is given two new labels. First, each node N is labeled with the time $t_p(L)$ for the literal L at N. Second, each edge (N, M) is labeled with $t_u(H)$, where H is the head literal of the clause C applied to produce node M. The program's all-solutions sequential execution time is the sum of the all t_p's and t_u's in the tree's processed region[2].

[2]Predicates such as cut may prevent traversal of parts of the tree.

```
fib(0,1).
fib(1,1).
fib(N,F) :-
    N > 1,
    N1 is N - 1,
    fib(N1,F1),
    N2 is N - 2,
    fib(N2,F2),
    F is F1 + F2.
```

Figure 1: A program to be timed

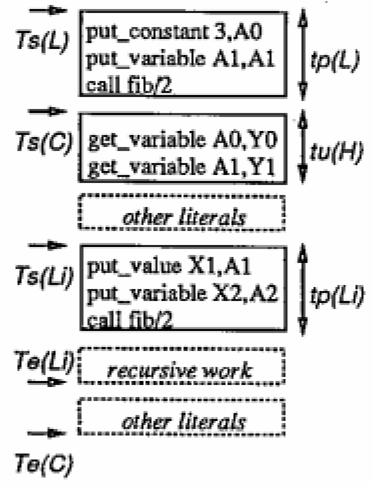

Figure 2: Execution of a timed literal L

2.1 Pure Prolog

Finding the minimum OR parallel time requires finding the *critical path* in the OR tree. For a *pure* Prolog program this is done by summing the t_p's and t_u's from the root to each leaf. The critical path has the largest such sum. Programs containing builtins such as `read`, `setof`, `recorda`, and `assert` require timing in sequential order. We first describe the method for pure programs and extend it to handle these predicates below.

Figure 1 shows a program to be instrumented, and Figure 2 shows its execution. The time at which literal L is to be processed is denoted by $T_s(L)$. If L is at the root of the OR tree, then $T_s(L) = 0$. Otherwise $T_s(L)$ is the time the preceding computation finished. Execution of L begins with the puts and `call`, which take time $t_p(L)$, as we noted above. Thus the earliest time any clause can be tried for L is $T_s(L) + t_p(L)$. This is the *start time* $T_s(C)$ for every clause C applied for L, since all are tried in OR parallel. Head unification for

C begins at $T_s(C)$ and is done by **get** (and **unify**) instructions. If successful, this completes at time $T_s(C) + t_u(H)$. If C is a fact, then the end time is $T_e(C) + 1$, where the 1 is for the **proceed** instruction.

If C is a rule, each literal L_i is processed as L was, begins at time $T_s(L_i)$, and ends successful execution at time $T_e(L_i)$. The first body literal begins at time $T_s(L_1) = T_s(C)+t_u(H)$. If the call from L_i is successful and returns at time $T_e(L_i)$, then the next literal L_{i+1} starts at time $T_s(L_{i+1}) = T_e(L_i)$. This continues until the last literal L_n completes at time $T_e(L_n)$, which is also the finish time $T_e(C)$ for C.

The time for a success is $T_e(L)$ for the last literal L in the top-level query. The time for a failed instruction in C is $T_s(C)$ plus the portion of $t_u(H)$ computed before the failure. Most builtins are given a cost of one, and builtin failure takes the same time as a successful call does.

The system maintains a global critical path time $T_{\max}$. Whenever a library routine performing head unification fails at time T_f, it examines $T_{\max}$, and stores the larger of the two times as the new $T_{\max}$. The library routine that computes $T_s(C)$ also updates $T_{\max}$, and the top-level query is modified to update it as well.

Figure 3 shows the timed version of Figure 1. Each clause has two new arguments, **Ts** and **Te**, and head unification is performed by routines such as **get_constant** and **get_variable**. These routines perform the corresponding WAM operation and update the critical path time. The first two clauses are facts, so the end time is computed by an **update_time** literal for the **proceed** instruction.

The third clause is a rule, so each body literal L has a preceding **update_time** literal. If L refers to a user-defined predicate this literal computes $T_s(L) + t_p(L)$ for use as the start time for the call. If L refers to a builtin predicates (except those in Section 4) the **update_time** literal adds $t_p(L)$, plus one for the builtin's execution time, and uses this as the *end time* for L.

Each clause also has an initial *index* literal that enables last call optimization. Moving head unifications to the body made indexing impossible, so this literal is added to perform first argument indexing. If this is not done, last call optimization rarely works. This literal appears sufficient for last call optimization with the Sicstus compiler.

```
fib(A,B,Ts,Te) :-
    (A == 0 ; var(A)),
    get_constant(A,0, Ts, Tu1),
    get_constant(B,1, Tu1, Tu2),
    update_time(Tu2, 1, Te).

fib(A,B,Ts,Te) :-
    (A == 1 ; var(A)),
    get_constant(A,1, Ts, Tu1),
    get_constant(B,1, Tu1, Tu2),
    update_time(Tu2, 1, Te).

fib(A,B,Ts,Te) :-
    get_variable(A,N, Ts, Tu1),
    get_variable(B,F, Tu1, Tneck),
    update_time(Tneck, 4, Te1),
    N > 1,
    update_time(Te1, 6, Te2),
    N1 is N - 1,
    update_time(Te2, 3, Ts3),
    fib(N1, F1, Ts3, Te3),
    update_time(Te3, 6, Te4),
    N2 is N - 2,
    update_time(Te4, 3, Ts5),
    fib(N2, F2, Ts5, Te5),
    update_time(Te5, 6, Te),
    F is F1 + F2.
```

Figure 3: Program after instrumentation

3 Adding AND parallelism

The critical path time determines the best possible OR parallel execution time. Often segments of a branch can execute simultaneously, and doing so would reduce that critical path time. This is AND parallel execution, and unlike OR parallelism, it requires testing for dependences even in pure Prolog programs. In this section we describe the application of Kumar's [1988] techniques for Fortran to estimate the best AND/OR parallel execution time. The method we describe extends his work to deal with the dynamic data structures and aliasing present in Prolog. We believe this framework has the advantage over other methods [Shen 1986, Tick 1987] of allowing us to extend it to measure critical path times in programs with user parallelism.

A program's dependences can only be exactly determined at execution time, since one execution may have a dependence while another does not. A compiler, to ensure legal execution, must assume a

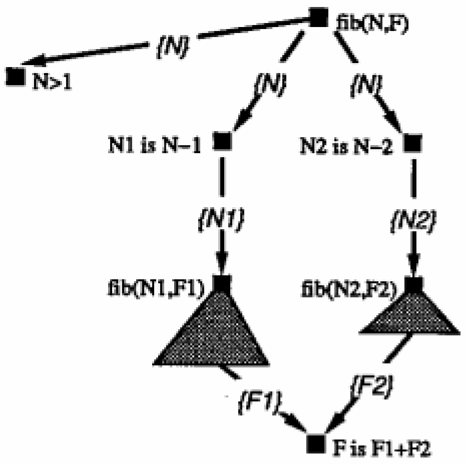

Figure 4: A dependence graph

dependence exists unless it can be proven not to. Because of this, compilers often infer many more dependences than are actually present in the program. Another use of the method we propose is to compute exact dependences to test the effectiveness of dependence tests.

There are a number of AND parallel execution models that differ in their treatment of the dynamic nature of dependences. The approaches range from dependence graphs that are static [Kalé 1987, Chang *et al* 1985, Wise 1986] to partly dynamic (conditional) [DeGroot 1984, Hermenegildo 1988] to completely dynamic [Conery and Kibler 1985]. Kal'e [1984] notes that in some rare situations it may be beneficial to evaluate dependent literals in parallel. His Reduce-Or Process Model allows for dependent AND parallelism, but his implementation [Ramkumar and Kalé 1989] supports only independent AND parallelism. Epilog [Wise 1986] also permits dependent AND parallelism, but provides a primitive (`CAND`) to curtail it. The model we have developed includes dynamic, independent AND parallelism, with a strict sequential ordering on dependent literals. We are only able to present the results here for independent AND parallel execution, though, because of a problem in the Prolog system used to execute the instrumented programs. In the future we hope to report the timings for the more general approach.

3.1 Dependences

The third clause in Figure 1 contains six body literals that might potentially execute in parallel. The arguments of the > builtin must both be numeric expressions, so to execute correctly the argument `N` to `fib` must be an integer. Because neither writes `N`, the two `is` goals can execute independently. Each reads `N` and produces a binding for `N1` or `N2`, the values of `N` for the recursive instances. Since all `fib` clauses read `N`, the recursive calls can only begin after their corresponding `is`. The final `is` literal requires the value of both `F1` and `F2`, so the two `fib` calls must precede the final `is`. There need be no other ordering between literals.

Figure 4 shows the *dependence graph* for the clause. There is a node for the initial call to `fib` and a node for each body literal. Recursive computations are represented by shaded areas. An arc between two nodes represents a *dependence*, or that the node at the tail must precede the node at the head of the arc. Dependence arcs are labeled with the variables causing them. Such a variable v causes a dependence δ in one of two ways. First, if the node at the tail of δ binds v, and and v is read at the head, then there is a *data dependence*. Second, if the node at the head of δ binds v and the node at the tail reads v using a metalogical predicate (`var`, `write`, etc.), then there is an *anti-dependence*. Anti-dependences arise when a literal succeeds with a variable v unbound and would fail or produce incorrect output because v is subsequently bound.

3.2 Shadow terms

Dependences are detected at run time by *shadow terms*. Each term t has a shadow term $\psi(t)$ associated with it, which mirrors t's structure. The shadow of an atomic term is the atom a. The shadow term of a compound term $t = f(t_1, \ldots, t_n)$ is $\mathrm{s}(\psi(t_1), \ldots, \psi(t_n))$, where $\psi(t_i)$ is the shadow for t_i.

A variable must be bound for a dependence to exist, so the shadow term for a variable keeps the binding times for that variable (there can be multiple bindings, since some may be variable-to-variable). The shadow of an unbound variable is unbound. If v is bound to any term t at time T by a `get_variable` or `unify_variable` instruction, the shadow variable $\psi(v)$ is dereferenced and the final variable is bound to the structure $\mathrm{w}(\psi(t), T)$. The same operation is performed if v is bound to a non-variable term t by any other instruction. If v is bound to another variable v' by any other instruction at time T, an alias has been created. The two shadows reflect this by dereferencing both $\psi(v)$ and $\psi(v')$ and binding the final variables of both to the term $\mathrm{w}(\psi'(v), T)$, where $\psi'(v)$ is a new unbound

```
fib(A, B, Sa, Sb, Ts, Te) :-
    (A == 0 ; var(A)),
    get_constant(A,0, Sa, Ts, Tu1),
    get_constant(B,1, Sb, Tu1, Tu2),
    update_times(Tu2, 1, Te).

fib(A, B, Sa, Sb, Ts, Te) :-
    (A == 1 ; var(A)),
    get_constant(A,1, Sa, Ts, Tu1),
    get_constant(B,1, Sb, Tu1, Tu2),
    update_times(Tu2, 1, Te).

fib(A, B, Sa, Sb, Ts, Te) :-
    get_variable(A,N, Sa, Sn, Ts, Tu1),
    get_variable(B,F, Sb, Sf, Tu1, Tu),
    max_shadow_time(Tu, [Sn], Tt1),
    update_time(Tt1, 4, Te1),
    N > 1,
    max_shadow_time(Tu, [Sn1,Sn], Tt2),
    update_time(Tt2, 6, Te2),
    N1 is N - 1,
    set_shadows([Sn1],[N1],Te2),
    update_time(Tu, 3, Ts3),
    fib(N1, F1, Sn1, Sf1, Ts3, Te3),
    max_shadow_time(Tu, [Sn2,Sn], Tt4),
    update_time(Tt4, 6, Te4),
    N2 is N - 2,
    set_shadows([Sn2], [N2], Te4),
    update_time(Tu, 3, Ts5),
    fib(N2, F2, Sn2, Sf2, Ts5, Te5),
    max_shadow_time(Tu, [Sf,Sf1,Sf2], Tt6),
    update_time(Tt6, 6, Te6),
    F is F1+F2,
    set_shadows([Sf], [F], Te6),
    max([Te1,Te2,Te3,Te4,Te5,Te6], Te).
```

Figure 5: AND/OR instrumented program

variable. If v is examined by a meta-logical builtin at time T, $\psi(v)$ is dereferenced, and the final variable is bound to $m(\psi'(v), T)$, where $\psi'(v)$ is a new unbound variable.

3.3 Dependences with shadow terms

Figure 5 shows `fib` after instrumentation for AND/OR parallelism. Each variable `V` in a clause has a shadow variable `Sv`, and each head argument has a shadow argument. The end time for a clause is the largest end time for any literal in that clause, as if each literal starts immediately after head unification and suspends until its dependences are satisfied. In Figure 5 the end time is shown as computed by a `max` literal at the end of the clause. This is for clarity of presentation only, because this would inhibit last call optimization. In the real version a current maximum is passed to each body literal in succession.

The head unification routines now include shadow variables as arguments, since it is in these instructions that dependences in user-defined predicates are enforced. These routines previously computed their finish time only from the start time and the cost of the instruction. Now there is the possibility that the instruction must wait until the shadow time for a variable causing a dependence before performing the unification. Hence the completion time is computed by performing the unification and keeping a current time. Whenever a term is referenced the current time becomes the maximum of the current time and the timestamp. The unification is then performed and the current time incremented.

Two other predicates enforce dependences involving builtins. The first, `max_shadow_time`, computes the earliest time the builtin's arguments are available[3] from the latest time in the arguments' shadows. This enforces data dependences that have the builtin as their head. The builtin's end time is computed by `update_time`, as before. The second predicate, `set_shadows`, builds shadows for changes to the arguments of a builtin. Shadows are built for those arguments that are bound or are examined by meta-logicals, and they are constructed from the variable bindings after execution. This handles builtins at the tail of a dependence. For some builtins such as `=..` this can be fairly complex.

4 Builtin predicates

Prolog has several types of builtin predicates, each with a different set of effects on critical path timing. We have already noted that meta-logical builtins (`var`, `write`, etc.) can cause anti-dependences. In this section we describe four other kinds of predicates and methods for timing each of them.

4.1 Predicates involving call

There are four predicates that implicitly use the meta-logical builtin `call`. They are `bagof`, `setof`, `not`, and `\+`. Timing these predicates requires two

[3]This predicate is also used to enforce independent-AND parallel execution, by making every user predicate strict

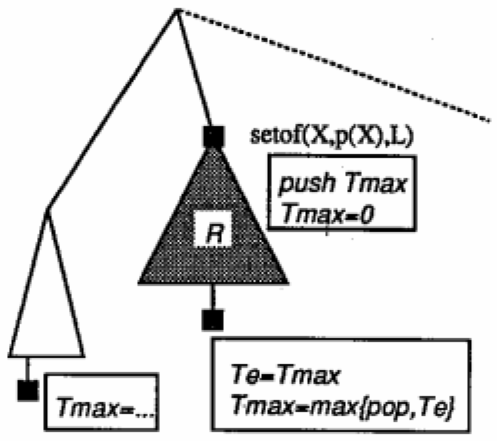

Figure 6: Processing setof

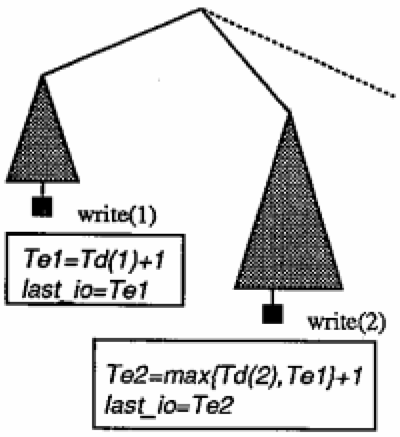

Figure 7: Processing the input/output predicates

kinds of special handling. First, since `call`'s arguments may be constructed at run time, instrumentation is done at run time. This is done by including the the instrumentation program in the timed program. Second, `setof`, `bagof`, `not` and `\+` traverse an entire OR tree, so their finish times are related to the longest path in that tree. A stack of maximum times is used with nested calls to these predicates to collect a subtree's maximum time. For `setof` and `bagof` we also add one for each solution for the cost of building the returned list.

Figure 6 shows the processing of a call to `setof` that computes all the solutions for the `p(X)` in region R and collects them in a list `L`. Since it traverses the whole OR tree R required to compute `p(X)`, `setof`'s finish time is the longest completion time in R. The maximum time is maintained by `update_time` in the global variable `Tmax`[4]. Since there may be a previous maximum time greater than the largest completion time in R, `Tmax` is pushed on a stack and the start time for the `setof` is used as `Tmax`. R is traversed and the maximum time is stored in `Tmax`, as always. The return time for `setof`, `Te` is `Tmax`. At the end of `setof` `Tmax` is set to the maximum of the stack value and `Te`, so again `Tmax` contains the global largest time.

4.2 Read and write

Neither `setof` nor pure Prolog cause dependences between branches in the OR tree. The input/output predicates (`read`, `write`, etc.) cause cross-branch dependences, since the observable order of input/output needs to conform to Prolog's left-to-right order. Figure 7 depicts the execution of a program with two `writes`, w_1 and w_2. Data dependence would permit each `write` to start when its arguments were ready (times $T_d(1)$ and $T_d(2)$ respectively) were it not for the order of output. w_1 must write its output before w_2, so to determine when input or output can be done we maintain a global variable `last_io`. In this example, w_2 cannot write its output until $\max\{T_d(2), \texttt{last\_io}\}$. Writes cost one time unit, so w_2 can start no earlier than $\max\{T)d(2), T_d(1)+1\}$. In the instrumented version each input/output predicate is preceded by a literal that updates `last_io`.

4.3 Recorda and recorded

Prolog also has the builtins `recorda`, `recorded`, and `erase` to manipulate an internal database. Parallel accesses to relations in the database must appear to preserve the sequential execution order. Accesses to different database relations do not affect one another, so this order is only within a relation. It is not necessary to serialize accesses to each relation to preserve the appearance of sequential access order. All we need is to guarantee that read accesses to an element by `recorded` occur after the write access that placed that element there, and that write accesses (`recordas` and `erases`) are ordered. The former is enforced by pairing each item placed in the database with its insertion time. Accesses by `recorded` wait until the maximum of the data dependence time T_d and the element's insertion time. The write order is enforced by labeling each relation with a `last_modify` that is updated

[4] In the implementation of our system the maximum time, along with a parallelism histogram, is maintained by several C routines accessed through a foreign function interface, but this is done only for the sake of efficiency.

Program Name	Serial WAM Instr.	OR Parallel Speedup	AND/OR Parallel Speedup
chat_parser	1014791	257	1596
crypt	31787	58	114
divide10	207	1	2
fast_mu	8899	9.1	10.7
flatten	5218	1.25	2.37
log10	119	1	1.2
meta_qsort	38675	2.1	3.7
mu	5925	16.7	17.7
nand	180145	5.4	14.3
nreverse	4460	1	1
ops8	163	1.04	2.8
poly10	307177	1.1	76.3
prover	7159	4.5	14.2
qsort	5770	1.3	1.5
queens8	33821	26.4	69.3
query	17271	243	480
reducer	279220	2	3.3
serialise	3199	1.4	1.9
tak	1431202	1.1	686
times10	207	1	1.9
unify	29490	1.6	3.5
zebra	261858	453	482

Table 1: Instrumented benchmark times

just like `last_io`.

4.4 Assert and retract

Prolog also allows `assert` and `retract` to modify the program at run time. These predicates are timed by the method for `call` and that for the internal database. The former is because the `asserted` clause can be constructed at run time, and hence the instrumentation must be done then. The latter is because predicates modified at run time must obey the access rules for database updates. The write-write (`assert` and `retract`) order is enforced by updating the `last_modify` for the predicate. The read-write ordering is maintained by adding a first literal to each `asserted` clause that records when it was added. This is used to determine the earliest time a read (a `clause` builtin or call to the modified predicate) can execute.

5 Analysis of programs

Table 1 presents the results obtained by instrumenting 23 of the University of California at Berkeley's UCB benchmarks. These programs range over a variety of sizes and purposes. There are several interesting facts to observe from these programs. First, David H. D. Warren's assertion [Warren 1987] that OR parallelism was likely to produce significant speedups on a range of programs appears to be borne out. Several programs achieved small speedups from OR parallelism, mostly due to shallow backtracking (e.g flatten, ops8, poly10, qsort, tak, unify). Improved indexing would probably eliminate most of this OR parallelism. A number of programs exhibited essentially no OR parallelism (e.g. divide10, log10, nreverse, times10).

In general, independent AND parallel execution improved the performance of programs already speeded up by OR parallel execution by a small factor (1-6). These programs have all shown reasonable speedups in real OR parallel systems[Szeredi 1989]. Our results show that there is plenty of parallelism in several of these programs to extend to much larger machines (e.g. consider chat_parser, query and zebra). Those with smaller speedups may profit from the introduction of independent AND parallelism.

Of the programs that were mostly OR-sequential, the majority get very small speedup by applying independent AND parallel execution. For divide10, log10, and times10, this is because the AND parallel sub-problems are very unbalanced; that is, one sub-problem is much larger than the other. For nreverse, the reason is that independent AND parallel execution is not able to execute the two body goals of nreverse in parallel. It is a recurrence, and is hence completely sequential. This can be addressed by replacing the algorithm or applying a parallel recurrence solver.

The best results for independent AND parallelism come from poly10 and tak. In both cases these give rise to fairly large numbers of independent subcomputations. In the case of tak, the branching factor is approximately three and the calling depth is large, so a large speedup is obtained. Qsort on a well-chosen input list with a better partition routine should be able to obtain similar results.

These results are just the beginning of understanding the parallelizability of programs, as we would like information on the more general AND and other sorts of parallelism. However, they can tell us something about how much speedup we can reasonably expect from parallel models. Moreover, examining these programs to see where dependences occur should help in designing restructuring transformations.

6 Conclusions

The amount of OR and AND/OR parallelism in a Prolog program can be effectively measured by sequentially executing an instrumented version of that program. The timings obtained this way give a best-possible speedup under two different parallelism models, and can be used for a number of purposes. First, they can be used to evaluate the ability of a parallel execution model to exploit parallelism. These results can suggest areas of improvement for such models. We intend to instrument a number of programs for this purpose.

With some relatively simple extensions this technique can measure the amount of a number of lower-level program characteristics. Among these are unification parallelism, backtracking properties, aliasing, data dependences, and dereference costs.

Prolog can also be extended with predicates for source-level parallelism. With proper timing methods, this instrumentation method can be used to evaluate restructuring transformations for Prolog. The instrumentation system we described has been extended with such predicates and we have begun to evaluate transformations. In the future we will describe these extensions to the instrumentation method as well as the results of our restructuring transformations.

Acknowledgments

The authors would like to thank David Padua for his many useful suggestions about this work.

References

[Ali 1990] Khayri Ali. The muse or-parallel prolog model and its performance. In *Proceedings of the 1990 North American Logic Programming Conference*, pages 757–776, 1990.

[Chang *et al* 1985] J. Chang, A. M. Despain, and D. DeGroot. And-parallelism of logic programs based on a static data dependency analysis. In *Proceedings of Compcon 85*, 1985.

[Conery and Kibler 1985] J.S. Conery and D.F. Kibler. And parallelism and nondeterminism in logic programs. *New Generation Computing*, 3:43–70, 1985.

[DeGroot 1984] D. DeGroot. Restricted and-parallelism. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 471–478. North Holland, 1984.

[Hermenegildo 1988] M. V. Hermenegildo. *Independent AND-Parallel Prolog and its Architecture.* Kluwer Academic Publishers, 1988.

[Kalé 1984] Laxmikant V. Kalé. *Parallel Architectures for Problem Solving.* PhD thesis, State University of New York at Stony Brook, 1985.

[Kalé 1987] Laxmikant V. Kalé. Parallel execution of logic programs: the reduce-or process model. In *Proceedings of the International Conference on Logic Programming*, pages 616–632, May 1987.

[Kumar 1988] Manoj Kumar. Measuring parallelism in computation intensive scientific/engineering applications. *IEEE Transactions on Computers*, 37(9), September 1988.

[Ramkumar and Kalé 1989] B. Ramkumar and L.V. Kalé. Compiled execution of the reduce-or process model on multiprocessors. In *Proceedings of the 1989 North American Conference on Logic Programming*, pages 313–331, October 1989.

[Shen 1986] Kish Shen. An investigation of the argonne model of or-parallel prolog. Master's thesis, University of Manchester, 1986.

[Szeredi 1989] Peter Szeredi. Performance analysis of the aurora or-parallel prolog system. In *Proceedings of the 1989 North American Conference on Logic Programming*, pages 713–732, 1989.

[Tick 1987] Evan Tick. *Studies in Prolog Architectures.* PhD thesis, Stanford University, June 1987.

[Warren 1983] David H. D. Warren. An abstract prolog instruction set. Technical report, SRI International, October 1983. Technical Note 309.

[Warren 1987] David H.D. Warren. The sri model for or parallel execution of prolog — abstract design and implementation. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–103. September 1987.

[Wise 1986] Michael Wise. *Prolog Multiprocessors.* Prentice-Hall International Publishers, 1986.