The Performance of Parallel Prolog Programs

Barry S. Fagin, *Member, IEEE,* and Alvin M. Despain, *Member, IEEE*

Abstract—**This paper presents performance results for a parallel execution model for Prolog that supports AND-parallelism, OR-parallelism, and intelligent backtracking. The results show that restricted AND-parallelism is of limited benefit for small programs, but produced speedups from 7 to 10 on two large programs. OR-parallelism was generally not found to be useful for the benchmarks examined if the semantics of Prolog were preserved.**

Of particular interest is the phenomenon of supermultiplicative behavior, in which the performance improvement obtained when more than one technique is employed is greater than the product of the performance improvements due to each technique individually. The implications of the performance results for parallel Prolog systems are discussed, and directions for future work are indicated.

Index Terms—**AND-parallelism, intelligent backtracking, logic programming, OR-parallelism, parallel symbolic computing, Prolog, supermultiplicative phenomena.**

I. Introduction

THIS work presents some performance results of a new execution model for parallel Prolog: the Parallel Prolog Processor model, or PPP. The execution model and its underlying architecture were developed as part of the Aquarius Project at the University of California at Berkeley. The Aquarious Project is a research investigation into the underlying issues of high performance symbolic computation, and in particular high performance execution of Prolog.

This paper attempts to shed new light on the behavior of parallel logic programs. Many execution models for parallel logic programming have been described in the literature. These include Conery's AND/OR process model [9], a general OR-parallel execution model [2], Li and Martin's Sync model [24], various dataflow models [1], [20], the Japanese Parallel Inference Engine [26], and the restricted AND-parallel execution model of Hermenegildo [22]. These models are theoretically powerful, but little is known about how these models would behave on actual programming tasks. Simulation results are extremely rare, and those studies that do exist are based on extremely small, "toy" programs [10], [22]. As a result, we know very little about the utility of parallel logic programming on realistic programs. This paper presents a detailed simulation study of a parallel Prolog execution model on a variety of benchmark programs, including a theorem prover and a Prolog compiler.

Our results indicate that for small programs restricted AND-parallelism is not effective in improving performance; even with 32 processors these programs ran no more than twice as fast. Better results are obtained with larger programs like the Boyer–Moore theorem prover and the Berkeley PLM complier; these programs ran 7 and 10 times faster, respectively. The experiments also show that OR-parallelism is not particularly effective for single-solution benchmarks. While the "mu_math" and "queens" benchmarks showed speedup factors of 4 and 5, the remaining single-solution programs showed less than two. Finally, we discovered that Prolog programs could exhibit supermultiplicative behavior. In particular, the speedup obtained in the "ckt4" benchmark by combining AND-parallelism, OR-parallelism, and intelligent backtracking was 77% greater than the product of the speedups of each technique taken individually.

The preliminary results of this research were first presented at the 14th International Symposium on Computer Architecture [19]. Since then, the simplified timing assumptions underlying our simulation results have been replaced with more realistic ones. We have also enlarged the benchmark set to include the Boyer–Moore theorem prover and the VanRoy Prolog compiler. Further work has also shown that preserving the semantics of sequential Prolog can have strongly negative performance consequences; this is discussed briefly here. Finally, this paper includes an analysis of supermultiplicative speedup in Prolog programs not provided in [19].

This paper is divided into five sections. Section II briefly explains the type of concurrency inherent in Prolog. Section III presents the simulated results of the PPP execution model and architecture on an extensive set of Prolog benchmarks, while Section IV presents our conclusions. Section V suggests possibilities for future work.

II. Parallelism in Prolog

Throughout this section, the reader is assumed to be familiar with Prolog. Readers unfamiliar with the language are referred to [8] and [27] as excellent introductory texts.

There are four basic types of parallelism inherent in Prolog: AND-parallelism, OR-parallelism, stream parallelism, and unification parallelism.

A. AND-Parallelism

When Prolog unifies a goal literal with the head of a compound clause C, the literals in the body of C are then unified sequentially. It is also possible, however, to resolve these literals in parallel if sufficient computing resources are available. This type of parallel execution is called *AND-parallelism*: it is the parallel traversal of AND subtrees in the execution tree. An example of AND-parallelism is shown in Fig. 1.

The principal difficulty with AND-parallelism is the problem of *binding conflicts*. If AND-subtrees of the execution tree are executed in parallel, the resulting unification substitutions may conflict. For example, consider the program

$$f(X) :- a(X), b(X).$$
$$a(1).$$
$$b(2).$$

Manuscript received May 11, 1988; revised October 13, 1988. This work was supported in part by the Defense Advanced Research Projects Agency, Arpa Order 4871, and was monitored by Space and Naval Warfare Systems Command under Contract N00039-84-C-0089.

B. S. Fagin is with the Thayer School of Engineering, Dartmouth College, Hanover, NH 03755.

A. M. Despain is with the Department of Electrical Engineering Systems, University of Southern California, Los Angeles, CA 90089.

IEEE Log Number 9040079.

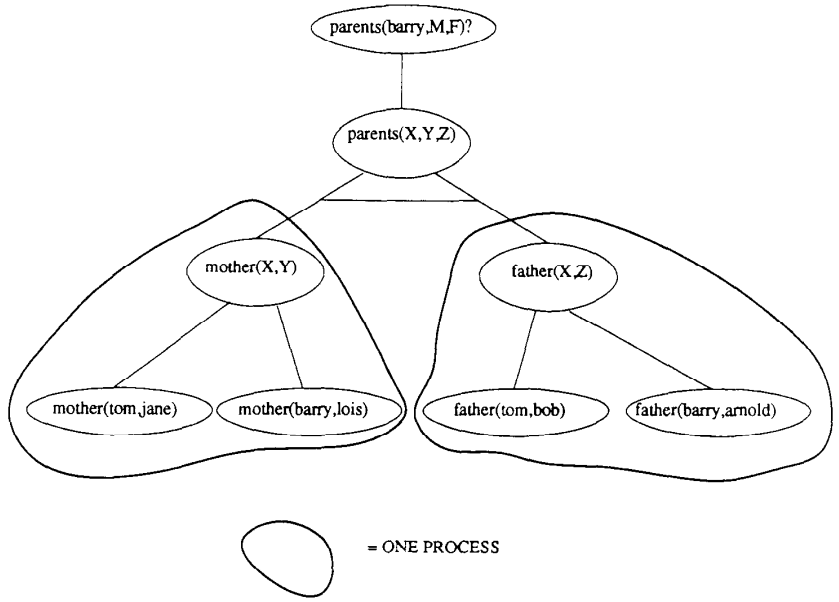


Fig. 1.

The literals $a(X)$ and $b(X)$ can theoretically be resolved in parallel. However, if both resolution steps are attempted concurrently, one will attempt to apply the unifying substitution $X = 1$, while the other will attempt $X = 2$.

AND-parallelism in which goals are allowed to execute in parallel even though they may bind shared variables is commonly referred to as *unrestricted* AND-parallelism. It requires some sort of synchronization for shared variables as well as a method of filtering sets of variable bindings returned from nondeterministic literals. (In the previous example, the set of substitutions $(X = 1, X = 2)$ and $(X = 2, X = 3)$ must be intersected to produce $X = 2$.) Because such schemes introduce considerable run-time overhead, most schemes for AND-parallelism incorporate program annotation to denote which goals produce and consume variable values [9], [12]. This information is then used to ensure that goals execute in AND-parallel fashion only if the resulting substitutions are conflict-free. This type of AND-parallelism is called *restricted* AND-parallelism [11]. The PPP execution model employs this type of AND-parallelism; variable binding conflicts are avoided through the use of compile-time analysis. The interested reader is referred to [4].

B. OR-Parallelism

In sequential Prolog, when a goal g with principal functor f is resolved by a Prolog program, unification is attempted with each clause in the procedure P for f. The clauses in P are examined from top to bottom. If unification with the head of a clause is successful, execution continues with the goals in the body, if any. If unification fails, the state of the machine is restored and the next clause in P is attempted.

However, given sufficient computing resources, it is also possible to perform the unification of g with all clauses in P and then execute the remaining subdirections associated with successful unifications in parallel. This type of concurrent execution is called *OR-parallelism*: it is the parallel traversal of OR subtrees in the execution tree of the program.

For example, in the following program

main :- $a(X), p(X)$.

$a(1)$:- b, c, d.
$a(2)$:- e, f, g.
$a(3)$:- h, i.

(other clauses for b, c, etc.)

main?

Example 1.0

the simultaneous unification of "$a(X)$" with "$a(1)$," "$a(2)$," and "$a(3)$," and the concurrent attempts to resolve their associated subgoals would be an example of OR-parallelism. This is shown below in Fig. 2.

OR-parallelism has been studied by numerous researchers [5], [2], [25], [10]. The chief difficulty with OR-parallelism is the problem of *multiple binding environments*. Consider the previous program. Each of the three children of $a(X)$ must see the variable X as unbound, and must be able to bind it to its own particular value, independent of the actions of the other children. Thus, even though the first child may have bound X to 1, the second child must not "see" that binding, instead being able to bind X to 2. Furthermore, there must be a way to transmit these bindings of X to the goal $p(X)$.

Most research in OR-parallelism execution of logic languages deals with ways to solve this problem. The PPP employs a technique called "hash windowing," first suggested by Borgwardt [2]. Early indications are that hash windowing outperforms other techniques [10], [6]. A detailed description of hash windowing in the PPP and the reasons for choosing it over other schemes is provided in [19].

C. Stream Parallelism

Stream parallelism has also been studied as a vehicle for concurrent computation [21], [13]. Stream parallelism in Prolog occurs when literals pass a "stream" of variable bindings to

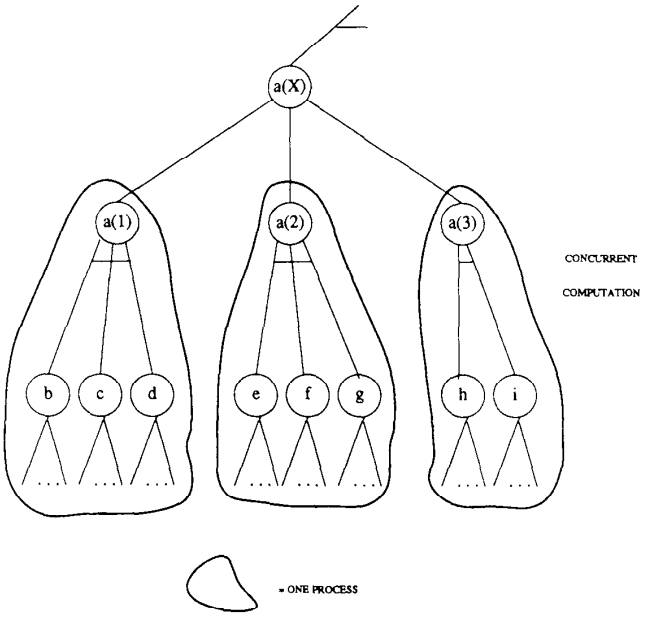


Fig. 2.

other literals, each of which is operated on concurrently. Literals that produce the bindings are called "producers"; literals that operate on the bindings generated by producers are called "consumers."

For example, consider the following program

main :- int(N), test(N), print(N).

int(0).
int(N) :- int(M), N is $M + 1$.

in which test(N) is any procedure that makes use of N. If, when int(N) succeeds, a new process is created that tests and prints N at the same time as further solutions for int(N) are generated, then such concurrent execution would be stream parallelism.

While at first glance stream parallelism may appear to be a new type of concurrency, in fact its effects can be duplicated using OR-parallelism. This is discussed in detail in [19].

D. Unification Parallelism

The last type of parallelism is unification parallelism. Unification parallelism is of a finer grain than the other types mentioned, occurring when the unification of multiple arguments in the head of a clause proceeds concurrently. For example, in the program

main :- $a(X,Y), \cdots$

$a(f(b,c),g(d))$.

the unification of X with $f(b,c)$ and Y with $g(d)$ could occur concurrently, using unification parallelism.[1]

[1]This contrasts with the definition of unification parallelism of Hwang *et al.* in [23], which defines unification parallelism as "the parallel matching of clauses in the Prolog database with the goal clause and the partial instantiation of variables to constant values." Hwang's definition is a subcase of OR-parallelism as we have defined it; other sources refer to it as "shallow" OR-parallelism [28]. Since our definition is corroborated by other sources (for example, [26]) we will continue to use unification parallelism as we have defined it.

While it is known that in the worst case unification is very unlikely to be sped up by parallel computation [16], most of the unification encountered in typical Prolog programs can be [7]. The addition of special hardware and the use of instruction scheduling to exploit unification parallelism is currently an active area of research. However, we note that unification parallelism occurs at the instruction set level, and not at the level of computing processes. Unification parallelism is of a much finer grain than AND or OR parallelism. Since this paper is concerned with the coarse-grained parallelism of Prolog, we will not discuss unification parallelism further.

III. Results

In this section, we present and discuss the performance of a parallel Prolog execution model, the PPP, on several benchmark programs. The PPP is an execution model for Parallel Prolog supporting restricted AND-parallelism and OR-parallelism. It is implemented through compilation to an abstract Prolog instruction set, a superset of the Berkeley PLM instruction set [17], [15]. A detailed description of the PPP execution model and instruction set is available in [18].

The results presented here represent the most comprehensive study of parallel Prolog known to the authors. We begin the section with a brief discussion of the modeled system and how the results were obtained. We then present the results of a 32-processor system on a large benchmark set, and discuss them in detail.

A. The Simulator

The results presented in this section were obtained with a multiprocessor simulator, written in C, and running under 4.3 BSD UNIX. The simulator is based on the level 1 PLM simulator developed by Tep Dobry [14], suitably modified to support the multiprocessing operations and new instructions of the PPP.

The simulator models an arbitrary number of processors connected to a single cycle memory system, as shown in Fig. 3.

Each processor is an extended version of the Berkeley PLM

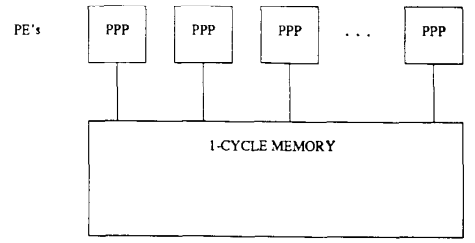


Fig. 3. The system under study.

TABLE I
THE BENCHMARKS

prog	# lines	description
divide10	13	symbolic differentiation
log10	13	"
ops8	13	"
times10	13	"
diff	19	"
palin25	24	index a list
qsd	20	quicksort 50 items
query	37	database query
con1	4	deterministic concat
con6	6	nondeterministic concat
mumath	28	simple theorem prover
queens	41	solve 4-queens problem
boresea	53	tests procedure calls
construct_list	28	tests list construction
construct_str	28	tests structure construction
deep_bak	25	tests deep backtracking
envir	19	tests environment allocation
map0	25	simple map coloring problem
match_list	31	tests list unification
match_str	38	tests structure unification
match_nested_str	24	tests nested structure unification
knight	27	knight's tour, 4 X 4 board
compiler	1597	prolog compiler
boyer	337	Boyer-Moore Theorem Prover
ckt4	35	designs simple circuit

Prolog processor [15]. The number of processors can be specified at run time, defaulting to 32.

Simulation is interleaved at the level of PPP instructions. That is, during a simulation run, processor 0 will be simulated for one PPP instruction, then processor 1, and so on. Processors that are not currently executing PPP code scan a process table for available work.

B. How the Speedup of the PPP is Reported

The speedup for a program P for N processors is obtained by dividing the execution time of P compiled for sequential execution by the execution time of P compiled for parallel execution on an N-processor system. Since a uniprocessor executing a program P compiled for parallel execution may run much slower than a uniprocessor executing P compiled for sequential execution, owing to the costs of concurrent processing, negative speedups will be exhibited when the costs of multiprocessing outweigh the benefits.

C. The Benchmarks

A total of 24 benchmarks were examined. All benchmarks were examined for opportunities to exploit AND-parallelism, OR-parallelism, and intelligent backtracking. The benchmarks used are shown below in Table I, along with their size and a brief description.

TABLE II
BENCHMARK SIZE (W-INSTRUCTIONS)

prog	seq	A	O	AO
divide10	198	214	x	x
log10	192	x	x	x
ops8	190	206	x	x
times10	198	214	x	x
diff	360	364	x	x
palin25	190	193	190	193
qsd	122	126	119	123
query	389	392	389	392
con1	31	x	x	x
con6	28	x	28	x
mumath	136	x	139	x
queens	214	216	203	227
boresea	216	220	x	x
construct_list	418	422	x	x
construct_str	518	508	x	x
deep_bak	85	89	85	89
envir	166	170	x	x
map0	144	x	144	x
match_list	528	532	x	x
match_str	726	730	x	x
match_nested_str	593	597	x	x
knight	387	x	393	x
compiler	10214	10222	x	x
boyer	2603	2607	x	x

prog	seq	A	O	I	AO	AI	OI
ckt4	281	285	285	292	289	294	296

One of our criticisms of previous work in parallel Prolog research was the small number of programs under study and their relatively trivial nature. In spite of this, our attempt to improve the situation has only been partially successful. While this work represents the largest study of parallel Prolog programs known to the authors, and while most of these benchmarks are larger than others previously studied, only a few of them can be regarded as "serious" programs. Nonetheless, we believe the results to be a significant first step.

None of these benchmarks use "assert" or "retract," Prolog predicates that modify the source program. The problems of side effects and self-modifying code in parallel execution are well known; a thorough treatment of them is beyond the scope of this paper. We suggest this topic as an area of future work.

The number of W-code instructions per benchmark are shown in Table II. This table lists the number of W-code instructions in the sequential version of the benchmark, as well as versions complied for AND-parallelism, OR-parallelism, and intelligent backtracking.

We note that only "ckt4" was able to take advantage of our semi-intelligent backtracking scheme, although opportunities for semi-intelligent backtracking may also exist in the compiler. This suggests that opportunities for semi-intelligent backtracking may be rare among Prolog programs.

D. Compilation Decisions in the PPP

Currently, programs are hand-compiled for simulation on the PPP simulator. Potential points for AND-parallel and OR-parallel execution are identified by the programmer, and then the sequential W-code is modified appropriately. We believe that the automatic detection of parallel execution points by a Prolog compiler is feasible, and suggest further investigation into this area. A simple algorithm for parallel execution point detection is given in [18].

TABLE III
EXPECTED SPEEDUP OF BENCHMARKS (32 PROCESSORS, NO OVERHEAD)

prog	A	O	AO
divide10	1.25	x	x
log10	x	x	x
ops8	1.81	x	x
times10	1.30	x	x
diff	2.69	x	x
palin25	1.11	1.28	1.30
qsd	1.74	1.24	1.44
query	1.05	24.98	1.13
con1	x	x	x
con6	x	2.52	x
mumath	x	6.72	x
queens	1.04	9.62	2.79
boresea	2.14	x	x
construct_list	8.61	x	x
construct_str	8.50	x	x
deep_bak	3.56	2.64	3.67
envir	10.83	x	x
map0	x	1.33	x
match_list	6.72	x	x
match_str	5.18	x	x
match_nested_str	5.32	x	x
knight	x	1.82	x
compiler	9.55	x	x
boyer	8.28	x	x

prog	A	O	I	AO	AI	OI	AOI
ckt4	1.13	1.79	8.48	1.93	12.39	19.29	28.72

E. Upper Bounds: Performance Under Optimal Conditions

To better understand the upper limits on the performance of the PPP, we first present the performance results of an "ideal" 32-processor PPP system below.[2] These results assume that all operations associated with multiprocessing take zero time: they represent upper bounds on the performance of the PPP.

The letters "*A*," "*O*," and "*I*" stand for AND-parallelism, OR-parallelism, and intelligent backtracking. The entries in a particular column represent the speedups obtained for benchmarks compiled using the techniques corresponding to the letters in the column. An "*x*" indicates that a benchmark could not exploit a particular technique.

We see immediately that a wide range of speedups are obtained. In some cases, a great deal of speedup is possible, while in other cases little improvement can be achieved. Before presenting more realistic performance measurements, we examine the benchmarks that showed relatively poor potential improvements (< 2) even under optimal conditions, in an effort to understand the limits of performance of parallel Prolog programs.

1) Poorly Performing Benchmarks: Benchmarks which showed no speedup of greater than 2 are shown below in Table IV.

2) Analysis: There are five factors limiting the performance of the programs in Table IV:

1) no parallel execution of goals possible
2) the position of solutions in execution tree
3) an unbalanced execution tree
4) no support for partial evaluation of shared data
5) the semantics of sequential Prolog.

[2]The present version of the simulator supports a maximum of 32 processes, so additional processors would not be effective.

TABLE IV
BENCHMARKS WITH SPEEDUP < 2

prog	A	O	AO
divide10	1.25	x	x
log10	x	x	x
ops8	1.81	x	x
times10	1.30	x	x
palin25	1.11	1.28	1.30
qsd	1.74	1.24	1.44
con1	x	x	x
map0	x	1.33	x
knight	x	1.82	x

TABLE V
FACTORS LIMITING BENCHMARK PERFORMANCE (32 PROCESSORS)

prog	factor
divide10	3
log10	1
ops8	3
times10	3
palin25	3, 4
qsd	3
con1	1
con6	3
map0	2, 5
knight	2, 5

The factor appropriate to each benchmark is shown in Table V.

We note that only the factors four and five are properties of the PPP: the other factors are properties of the benchmarks themselves.

a) Benchmarks with no inherent concurrency: We see in Table III that the log10 and con1 benchmarks utilized neither AND-parallelism nor OR-parallelism. An inspection of these benchmarks will show that at any point, only one goal exists that can be reduced. Thus, these benchmarks are inherently sequential, and cannot be sped up using parallel goal reduction.

b) Positioning of solutions in the execution tree: The map0 and knight benchmarks are quite similar; they perform an OR-parallel search of a problem space for a solution satisfying certain constraints. With OR-parallelism, the further to the right a solution is in the execution tree, the greater the benefits to be gained from OR-parallel computation. Solutions near the left correspond to those that would be found more quickly by the left-to-right traversal of sequential Prolog, and represent less of a performance win.

With the map0 and knight benchmarks, the desired solution lies very close to the leftmost path in the execution tree. Thus, even though many processes are created, the vast majority of them perform wasted work, and do not contribute to improved performance.

c) Benchmarks with unbalanced execution trees: The remaining benchmarks have potential for parallel execution that the PPP can recover, but do not exhibit significant performance improvement. This is due to the shape of the execution trees they construct. Typical examples of this are the divide10 and times10 benchmarks. If one examines the source code, it is immediately obvious that the calls to "*d*" in the bodies of the first four clauses can be reduced in parallel, exploiting AND-parallelism. However, the resulting performance improvement depends on the distribution of work involved in each reduction. If one reduction takes ten times as much time as the other, then

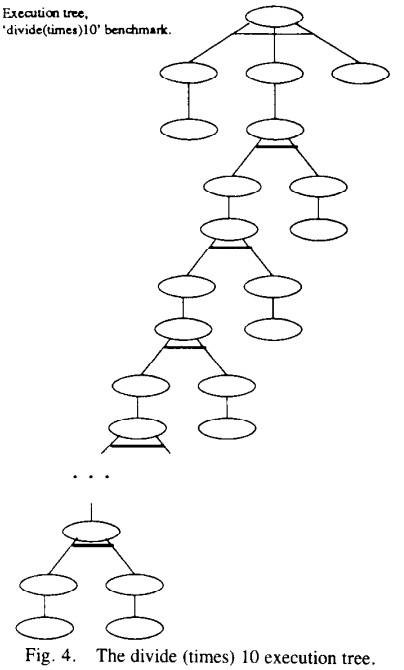


Fig. 4. The divide (times) 10 execution tree.

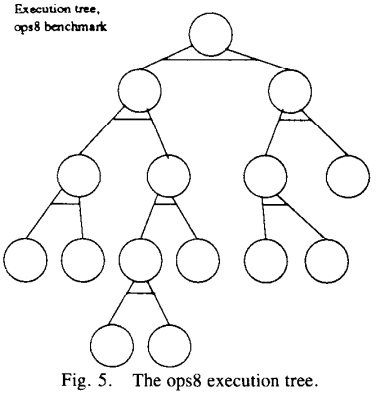


Fig. 5. The ops8 execution tree.

executing both in parallel will yield a performance improvement of at most 10%.

In fact, the skewed distribution of work is exactly what occurs in the times10 and divide10 benchmarks. Both these programs build the same execution tree, shown in Fig. 4. We can see that while the tree includes goals that are reduced in AND-parallel fashion, the resulting workload is extremely unbalanced; one goal reduces very quickly, while the other takes much more time. This is because the two subexpressions to be differentiated differ widely in complexity.

The ops8 benchmark, which fares a little better, does so because of a more balanced execution tree (Fig. 5).

d) The lack of support in the PPP for partial evaluation: One program, palin25, is coded in a sequential manner, but could conceivably be sped up through the partial evaluation of shared data. The main clause of the program can be viewed as a software pipeline:

$$\text{serialize}(L,R) :-$$
$$\text{pairlists}(L,R,A), \text{arrange}(A,T), \text{numbered}(T,1,N).$$

The PPP must execute these three goals sequentially; "arrange(A,T)" cannot execute until A is fully instantiated, just as "numbered($T,1,N$)" cannot execute until T is fully instantiated.

Other execution models, however, such as [2] permit lists to be processed as pipelined data structures; values are passed at the head of the list to consumer processes, while new values are placed at the tail. At present, the PPP does not support this type of data pipelining; hence, the amount of concurrency it can recover from this benchmark is limited to the use of AND-parallelism in the code for "arrange."

e) Preserving the semantics of sequential Prolog: Since the PPP is intended as a parallel execution model for Prolog, the semantics of sequential Prolog are preserved under the PPP execution model. Sequential Prolog simulates nondeterminism by trying the clauses of a procedure in top-to-bottom order, restoring state in between each attempt. The PPP can initiate these attempts in parallel, but the answers are returned in the order that sequential Prolog would return them. Under certain circumstances, this can be quite costly.

Suppose, for example, that a problem space is being searched by OR-processes, and that one of the processes finds a solution quickly while other processes are still executing (Fig. 6). Because the PPP preserves the left-to-right ordering of sequential Prolog, it must wait until processes to the left of the succeeding process have terminated before discovering the solution. This can have adverse effects on performance, as we will see shortly.

f) Shallow versus deep OR-Parallelism: Numerical researchers, including Hwang [23] and Syre [28], have distinguished between the OR-parallel unification of clause heads and the subsequent OR-parallel processing of body goals. These two types of OR-parallelism are referred to as "shallow" and "deep" OR-parallelism. The performance results of the qsd and palin25 benchmarks indicate that shallow OR-parallelism, even under ideal conditions, will produce only a slight performance improvement.

The qsd and palin25 benchmarks each contained a procedure compiled for shallow OR-parallelism. These are shown below:

partition procedure, qsd benchmark

$$\text{partition}([X \mid L],Y,[X \mid L1],L2) :- X < Y,!,\text{partition}(L,Y,L1,L2).$$
$$\text{partition}([X \mid L],Y,L1,[X \mid L2]) :- \text{partition}(L,Y,L1,L2).$$
$$\text{partition}([],\_,[],[]).$$

split procedure, palin25 benchmark

$$\text{split}([X \mid L],X,L1,L2) :- !,\text{split}(L,X,L1,L2).$$
$$\text{split}([X \mid L],Y,[X \mid L1],L2) :- \text{before}(X,Y),!,\text{split}(L,Y,L1,L2).$$
$$\text{split}([X \mid L],Y,L1,[X \mid L2]) :- \text{before}(Y,X),!,\text{split}(L,Y,L1,L2).$$
$$\text{split}([],\_,[],[]).$$

We see with these two procedures that even though they are deterministic, some OR-parallel execution is still possible through multiple head unification. In addition, the goal "before" can be executed in both clauses for "split," making the OR-parallelism a little "deeper." However, we saw in Table III that the employing of OR-parallelism for these procedures led to very small improvements in performance even under optimal conditions. This is because no performance is gained if the first clause is chosen, and even if other clauses are chosen a fast sequential

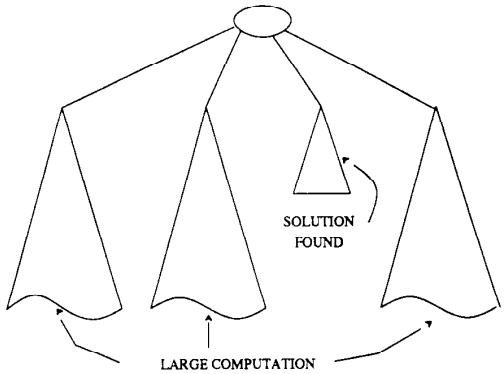


Fig. 6. Computational asymmetry in an OR-parallel search tree.

TABLE VI
BENCHMARKS WITH SPEEDUP > 2

prog	A	O	AO
diff	2.69	x	x
query	1.04	24.98	1.13
mumath	x	6.72	x
queens	1.04	9.62	2.79
boresea	2.14	x	x
deep_bak	3.56	2.64	3.67
construct_list	8.61	x	x
construct_str	8.50	x	x
envir	10.83	x	x
match_list	6.72	x	x
match_str	5.18	x	x
match_nested_str	5.32	x	x
compiler	9.55	x	x
boyer	8.28	x	x

prog	A	O	I	AO	AI	OI	AOI
ckt4	1.13	1.79	8.48	1.93	12.39	19.29	28.72

Prolog system like the PLM can restore state in between clauses in approximately 20 microcycles, incurring little performance penalty. This suggests that the utilization of shallow OR-parallelism is not likely to be very effective. In addition, we will see that, at least in the PPP, shallow OR-parallelism severely degrades performance as the system becomes bogged down in execution overhead.

By contrast, the query and ckt4 benchmarks, which employed deep OR-parallelism, performed well, even in the presence of execution overhead. This indicates, as expected, that deep OR-parallelism is more likely to yield significant performance improvement than shallow OR-parallelism.

3) Benchmarks with Higher Potential Speedup: Benchmarks with speedups of a factor of 2 or greater are shown in Table VI.

4) Analysis: We see that the last eight benchmarks in the first table all showed speedups greater than five. This is because these programs create tasks that perform large amounts of work. This suggests that compilers for parallel logic programming systems should spend time determining which goals have large subtrees associated with them. These goals are good candidates for parallel execution.

We also note that unlike the other symbolic differentiation benchmarks, the "diff" program has a potential speedup of greater than 2. This is because diff differentiates four expressions of approximately equal complexity in parallel. The resulting computational load is well balanced, leading to improved performance.

TABLE VII
SUPERMULTIPLICATIVE PERFORMANCE IN THE CKT4 BENCHMARK (NO OVERHEAD)

A	O	I	product	actual speedup	% diff
1.13		8.48	9.58	12.39	+29%
	1.79	8.48	15.17	19.29	+27%
1.13	1.79	8.48	17.15	28.72	+67%

We also notice that the speedup obtained by combining AND and OR parallelism is always less than the product of the two techniques utilized separately. For all of these benchmarks, this effect is due to system saturation. By combining AND and OR parallelism, the maximum number of processes in the system is quickly reached, forcing sequential execution.

5) Supermultiplicative Performance Improvements: One of the most interesting phenomena present in Table VI is shown in the results of the "ckt4" program. This benchmark is unique among the programs examined in that it can take advantage of AND-parallelism, OR-parallelism, and intelligent backtracking. AND and OR parallelism yield a small bit of performance improvement, while intelligent backtracking yields a large improvement. However, the speedup obtained by combining intelligent backtracking with the other techniques is always *greater* than the product of the speedups when taken separately. Just how much "extra" performance can be obtained is shown in Table VII.

We refer to phenomena of this type as *supermultiplicative*. Supermultiplicative performance improvements are counterintuitive; one might wonder if such results are even possible, let alone how they arise. Fortunately, they are both possible and explainable.

Supermultiplicative performance is counterintuitive because our intuition treats speedup techniques as applying to an entire program. When we find that a technique A speeds up a program by a factor S_A, we assume that all parts of the program run S_A times faster. So when technique B speeds up a program by a factor of S_B, we naturally expect that both techniques will improve performance by a factor of $S_A * S_B$; we assume that the speedup techniques are *uncorrelated*.

In some cases, this view of performance is correct. Speedup techniques that are independent of the program, such as a shortened cycle time or the addition of a cache, can be expected to improve performance of all parts of a program equally, resulting in multiplicative performance improvement when combined. However, when techniques are considered that affect different parts of the program in different ways, this view is no longer valid. Suppose, for example, that the program under consideration consists of two parts, each taking time $T1$ and $T2$ to execute. Suppose furthermore that parts 1 and 2 can be sped up by factors of S_A and S_B using techniques A and B. (Technique A leaves part 2 unaffected, while technique B leaves part 1 unaffected.) If only one technique is employed, the performance is limited by the part of the program that the technique cannot improve. If both techniques are used, however, then both parts of the program run faster, and the resulting speedup can be substantially greater than the product of the speedups taken separately. This behavior is shown graphically in Fig. 7.

Suppose we have a program divided into two parts, each part taking 4 hours. Suppose further that the first part can be run in

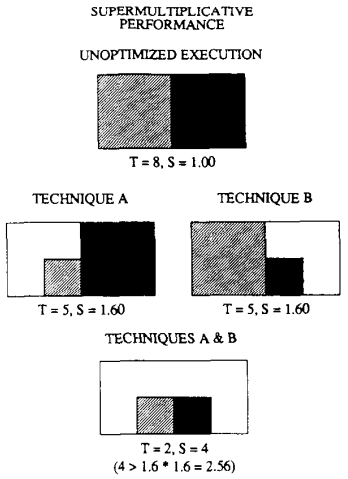


Fig. 7. A pictorial description of supermultiplicative performance.

one fourth the time using technique A, and the second in one fourth the time with technique B. If only technique A or B is used, the program takes 5 hours to run, giving a speedup of 1.6. If, however, both techniques are used, then the program takes only 2 hours. This gives a speedup of 4, as contrasted with $1.6 \times 1.6 = 2.56$. In this example, the speedup techniques are *correlated*; supermultiplicative interaction occurs.

This is exactly what happens with the ckt4 benchmark. Certain parts of the program can be sped up by using intelligent backtracking, others by using AND-parallelism, and others by using OR-parallelism. These techniques are correlated; by combining them, each part of the problem is sped up, and supermultiplicative performance improvement is observed. We propose a better understanding of the correlation of Prolog performance improvement techniques as a topic for future research.

F. More Realistic Performance Results

While the optimal results just presented are useful for understanding parallel Prolog, it is equally important to model the effects of execution overhead. As we have said earlier, most parallel execution models for Prolog are highly theoretical in nature, and tend to ignore the costs of multiprocessing.

In fact, many aspects of execution overhead need to be taken into account when modeling parallel systems. For one, process creation time is always nonzero. Loading and saving process context takes time, as does the implementation of a process scheduling algorithm. Typically, parallelizing a logic programming system introduces new operations that are required to maintain the integrity of the system, such as searching a binding window chain [10] or waiting to see if other processes have terminated [22]. These operations represent additional overhead that must be taken into account if realistic performance measurements are to be obtained.

To model the effects of these and other multiprocessing tasks, we have added code to the PPP simulator that simulates the PPP architecture executing these routines at the register transfer level. Using information obtained from the existing PLM processor [15], the time required to perform the various multiprocessing operations can be determined. For further details, the reader is referred to [18].

The performance results of the PPP on the previous benchmark set, taking into account the effects of parallel processing overhead, are shown in Tables VIII and IX.

TABLE VIII
EXPECTED SPEEDUP OF BENCHMARKS (32 PROCESSORS, OVERHEAD INCLUDED)

prog	A	O	AO
divide10	0.85	x	x
log10	x	x	x
ops8	1.35	x	x
times10	0.83	x	x
diff	2.53	x	x
palin25	1.11	0.38	0.45
qsd	1.65	0.38	0.41
query	1.01	17.33	0.69
con1	x	x	x
con6	x	0.62	x
hanoi	x	x	x
mumath	x	4.88	x
queens	0.57	6.75	1.48
boresea	1.46	x	x
construct_list	7.01	x	x
construct_str	6.90	x	x
deep_bak	2.69	0.48	1.10
envir	9.43	x	x
map0	x	0.48	x
match_list	5.40	x	x
match_str	4.43	x	x
match_nested_str	4.49	x	x
knight	x	1.62	x
compiler	9.53	x	x
boyer	7.09	x	x

prog	A	O	I	AO	AI	OI	AOI
ckt4	1.01	1.42	8.48	1.48	11.23	15.24	21.64

TABLE IX
DEGRADATION OF PERFORMANCE DUE TO OVERHEAD
DEGRADATION = $1 - (S_{REAL}/S_{OPT})$
(AN ENTRY OF 00* INDICATES A VALUE SLIGHTLY GREATER THAN 0)

prog	A	O	AO
divide10	.32	x	x
log10	x	x	x
ops8	.25	x	x
times10	.36	x	x
diff	.05	x	x
palin25	.02	.70	.65
qsd	.05	.69	.71
query	.04	.30	.39
con1	x	x	x
con6	x	.75	x
hanoi	x	x	x
mumath	x	.27	x
queens	.45	.30	.47
boresea	.32	x	x
construct_list	.19	x	x
construct_str	.19	x	x
deep_bak	.24	.82	.70
envir	.13	x	x
map0	x	.36	x
match_list	.20	x	x
match_str	.14	x	x
match_nested_str	.16	x	x
knight	x	.10	x
compiler	.00*	x	x
boyer	.14	x	x
avg	.18	.48	.58

prog	A	O	I	AO	AI	OI	AOI
ckt4	.11	.20	.00	.23	.15	.21	.25

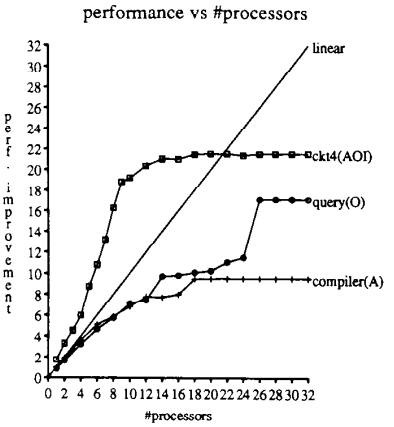


Fig. 8.

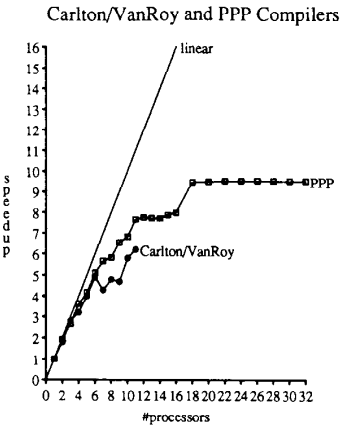


Fig. 9.

TABLE X
SUPERMULTIPLICATIVE PERFORMANCE IN THE CKT4 BENCHMARK (OVERHEAD INCLUDED)

A	O	I	product	actual speedup	% diff
1.01		8.48	8.56	11.23	31%
	1.42	8.48	12.04	15.24	26%
1.01	1.42	8.48	12.16	21.64	77%

1) Analysis: First, we note that the supermultiplicative performance improvements shown by the ckt4 benchmark remain even when execution overhead is taken into account (Table X). The improvement over the product of individual speedups is as good or better than the ideal case.

Referring back to Table VIII, we see that for the benchmarks that were compiled for shallow OR-parallelism, psd and palin25, performance dropped dramatically when overhead was modeled. This is because shallow OR-parallelism offers a small performance advantage to begin with, and is quickly overwhelmed by process creation and window management overhead.

We also see that the overhead associated with AND-parallelism is much less than that associated with OR-parallelism. This is as expected. For one, the exploitation of OR-parallelism can lead to an explosion in the number of processes created. In addition to this, OR-processes must manage multiple binding environments. These two factors increase execution overhead dramatically, contributing to the poorer performance of OR-parallel benchmarks.

2) Performance Versus # Processors: The benchmarks that performed the best using AND-parallelism and OR-parallelism were "compiler" and "query," respectively, while the only benchmark that made use of intelligent backtracking was "ckt4." The performance improvement of these benchmarks as a function of the number of processors, taking into account the effects of execution overhead, is shown in Fig. 8.[3]

The results for the compiler are similar to those reported by Carlton and Van Roy in [3]. In this paper, the authors report a peak speedup of 6.2 for 11 processors. For the PPP, a speedup of 7.6 was obtained with 11 processors, with a peak of 9.53 for 32. A plot of the PPP and Carlton/VanRoy curves for the PLM compiler is shown in Fig. 9.

We caution against a strict comparison, for two reasons. For one, Carlton and Van Roy employ a fixed number of processes per processor. No such restriction exists in the PPP simulator, in which processors can pick up any available process in the process table. Additionally, the program compiled by the PPP parallel compiler is different from that used by Carlton and Van Roy, which is too large for the PPP simulator. Nonetheless, the shape of their performance curve is quite close to that of the PPP, as are their performance values. This provides extremely strong evidence of the feasibility of fast compilation of Prolog on a multiprocessor.

Referring back to Fig. 9, the principal interesting feature of the query benchmark is the dramatic rise in performance at $P = 25$ processors. This occurs because the benchmark creates 25 processes. If fewer than 25 processors are available, execution time is limited by the processor(s) that must execute more than one process (recall that "query" generates all solutions to its top level goal). When the number of available processors matches the number of processes, performance improves dramatically. This suggests that a knowledge of the number of processors in the system will be helpful in the compilation of parallel logic programs, so that the compiler can attempt to match the number of processes to the number of processors.

The "ckt4" benchmark exhibits a performance improvement greater than the number of processors, due to intelligent backtracking. The performance of this benchmark when the effects of intelligent backtracking are removed is shown in Fig. 10.

This figure and Fig. 9 show pictorially what we have already demonstrated quantitatively: intelligent backtracking can interact supermultiplicatively with AND-parallelism and OR-parallelism to produce improved performance.

IV. CONCLUSIONS

We summarize the main conclusions of this paper below.

A. The Effectiveness of Restricted AND-Parallelism

The results indicate that for small programs, like the symbolic differentiation benchmarks, restricted AND-parallelism is not effective in improving performance; when execution overhead is taken into account these programs ran no more than twice as fast. Better results are obtained on large programs like the

[3]We use the term "performance improvement" instead of the more customary term "speedup" due to the use of intelligent backtracking in the ckt4 benchmark. Intelligent backtracking is an algorithm modification, so the strict definition of speedup does not apply to the "ckt4" benchmark. For the query and compiler benchmark, however, the results shown represent speedup.

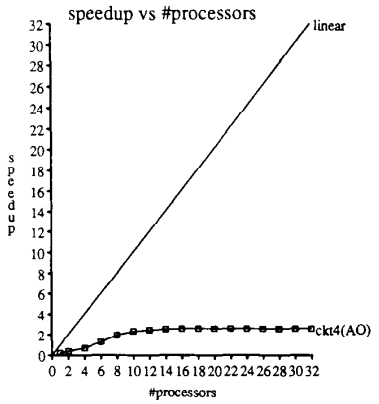


Fig. 10.

Boyer–Moore theorem prover and the Berkeley Prolog compiler.

It is difficult to develop a metric for the cost of parallelizing an execution model. We propose the following definition: let the optimal speedup of a benchmark be the speedup obtained on an infinite number of processors with no parallel execution overhead. Let the real speedup be the speedup obtained on an infinite number of processors with execution overhead taken into account. The cost of parallelizing an execution model for a particular benchmark is then defined to be the ratio of the real speedup to the optimal speedup, subtracted from one. Thus, an execution model that requires little overhead when parallelized has cost close to zero, while an execution model that requires large amounts of overhead to support parallel execution has a cost close to one. Note that this is just the derivation from optimality measure reported in Table IX, averaged over the benchmark set.

Based on this definition, the results indicate that restricted AND-parallelism in the PPP can be utilized at a cost of about 20% (see Table IX). Hermenegildo, in [22], estimates the cost of restricted AND-parallelism and in his execution model, the RAP-WAM, to be no greater than 14%. This figure is based on simulation studies of two programs, both of which were written solely to test the RAP-WAM model. Hermenegildo states quite clearly that the purpose of simulation in his thesis was solely to validate the model, and was not intended to be an exhaustive study. Since the PPP supports OR-parallelism in addition to AND-parallelism, and since the results presented here are based on studies of 25 preexisting Prolog programs, we believe that the PPP compares favorably to the RAP-WAM.

We see in Table IX that AND-parallelism improves the performance of the PLM compiler by a factor of ten, and speeds up the Boyer–Moore theorem prover by a factor of seven. These are significant performance improvements, on nontrivial, useful Prolog programs; we believe they should lay to rest any doubts about the potential utility of restricted AND-parallelism.

B. *OR-Parallelism and the Semantics of Prolog*

OR-parallelism comes at a higher cost than AND-parallelism, owing chiefly to the overhead of managing multiple binding environments. Thus, it should be utilized more carefully. In addition, the question of single-solution versus all-solution computation is extremely important.

If all solutions to a problem are desired, then OR-parallelism becomes equivalent to AND-parallelism in the sense that all descendants of a node represent tasks that must be carried out.

TABLE XI
THE EFFECT OF PRESERVING THE SEMANTICS OF SEQUENTIAL PROLOG

NORMALIZED EXECUTION TIMES			
prog	SEQ	PAR	PAR*
map	1.00	.74	.64
knight	1.00	.54	.11
mumath	1.00	.14	.11
ckt4	1.00	.55	.02

Unlike single-solution computation, all processors perform useful work, which leads to significant speedup. This is the case with the query benchmark, which yielded a performance improvement of 17. So for all-solution computation, OR-parallelism appears to be a useful technique to improve Prolog performance.

We believe, however, that single-solution computation occurs more often in Prolog programs, and will dominate as Prolog becomes more and more widely used. More likely than not, the user will not be interested in all possible chip designs, or all proofs of a theorem, but instead will seek a single solution that satisfies the constraints of the problem. For a single-solution computation, significant performance improvement was the exception rather than the rule. The "mu_math" and "queens" benchmarks showed a factors of four and five improvement, with the others showing factors of less than two.

The reason for the poor performance of OR-parallelism on single-solution benchmarks is the adherence of the PPP to the semantics of sequential Prolog. As we have pointed out previously, honoring the semantics of Prolog in an OR-parallel execution environment has the potential to severely inhibit performance.

In an attempt to measure the degree of performance inhibition caused by the semantics of sequential Prolog, four single-solution benchmarks were recompiled with a relaxed version of solution ordering. This was achieved by adding a "quit" instruction in the code, so that when any stream of computation executed it, even if other OR-siblings were still running, the entire program terminated. The results of the original and modified versions of the benchmarks are shown in Table XI. (The PAR column shows the original results, while the PAR* column shows the results with relaxed semantics.)

We see that even a slight change in the semantics of the PPP yields a dramatic performance improvement. We caution, however, against an improper interpretation of these figures. For example, one cannot divide the PAR* column into the SEQ column to obtain "speedup" figures comparable to those in Table III. Such a calculation is only meaningful when comparing two versions of the same algorithm, which is not the case with the PAR and PAR* results. Nonetheless, it seems evident that for single-solution computation, OR-parallelism is not likely to yield significant performance improvement unless the semantics of sequential Prolog are relaxed. We note that this is true regardless of the execution model used. All parallel execution models that preserve the semantics of sequential Prolog have this same problem: a solution to the original query may be found before searches of the branches to its left have terminated. Our results therefore indicate that for maximum potential performance improvement, the semantics of sequential Prolog should be discarded.

In spite of this conclusion, we feel the decision to adhere to the semantics of Prolog for the duration of the experiment was a valid one. If the possibility exists to achieve significant performance improvement without the design of a new language, then

such a possibility should be investigated. Indeed, the results indicate that through the utilization of restricted AND-parallelism, significant performance improvement of standard Prolog is achievable. However, the results also clearly show that the adherence to the semantics of sequential Prolog limit the utility of OR-parallelism, and that for further improvements modifications to Prolog are needed.

C. Costs and Benefits of Process Creation

The most important result shown in Tables III, VIII, and IX is the importance of cost/benefit analysis in parallel logic programming. The decision of whether or not to execute a goal in parallel *must* be made with regard to the costs and benefits of parallel computation. Simple visual inspection of a program is not sufficient. Many programs that at first glance contain the potential for parallel computation actually run slower when the costs of multiprocessing are taken into account. The actual costs of parallel processing will depend on the particular execution model under study, but we believe Tables III, VIII, and IX dramatically show the importance of a detailed consideration of the costs and benefits of parallel goal reduction in parallel logic programming systems. The construction of a cost/benefit model as an aid to compilation is proposed as a topic of future research.

D. Supermultiplicative Behavior in Logic Programs

Finally, we note that at least one program exhibits supermultiplicative behavior when AND-parallelism, OR-parallelism, and intelligent backtracking are combined. While we recognize that this is only one example, there is every likelihood that other programs not studied can exhibit similar behavior. This suggests that the effects of various performance enhancing techniques on logic programs should be considered *in toto*, and not in isolation from each other. If the techniques used map properly onto the bottlenecks of the problem, improvements greater than those attributable to individual techniques may be observed.

V. Future Work

The results of our work suggest several areas for future investigation. Since the decision of whether or not to execute a goal in parallel should be made according to a cost/benefit analysis, intelligent compilers should be developed for such a task. Side effects in parallel logic programs present difficult challenges that future research will have to address. We have also seen that the shape of the execution tree is an important factor in determining Prolog performance. This suggests that source-to-source program transformation may be useful in creating more efficient parallel logic programs. Further work remains to be done in analyzing the effect of memory system latency on the performance of parallel logic programs; clearly the single-cycle assumptions presented here should be modified to take into account bank conflicts, cache coherency issues, and so forth. Finally, we have already noted the possibility of supermultiplicative behavior in parallel logic programs. We believe this area merits considerable further study.

Acknowledgment

B. Fagin would like to thank his thesis readers, A. Despain, Y. Patt, and R. Solovay for their time and valuable comments. He is also grateful to the reviewers for their careful and constructive criticism. Both authors have benefited from interaction with Dr. V. P. Srini, Dr. T. Dobry, Dr. W.-M. Hwu, and Dr. W. Citrin, and the members of the Aquarius Research Group: P. Bitar, B. Bush, M. Carlton, C. Chen, R. David, J. Gee, B. Holmer, R. McGeer, S. Melvin, T. Nguyen, A. Singhal, M. Shebanow, J. Swensen, J. Tam, H. Touati, J. Wilson, and R. Yung.

References

[1] L. Bic, "A data-driven model for parallel interpretation of logic programs," in *Proc. Int. Conf. Fifth Generation Comput. Syst.*, 1984, pp. 517–523.

[2] P. Borgwardt, "Parallel Prolog stack segments on shared-memory multiprocessors," in *Proc. 1984 Symp. Logic Programming*, Feb. 1984, pp. 2–11.

[3] M. Carlton and P. V. Roy, "A distributed Prolog system with AND-parallelism," in *Proc. Hawaii Int. Conf. Syst. Sci. '88*, Honolulu, Hawaii, Jan. 1988, submitted for publication.

[4] J. H. Chang and A. M. Despain, "Semi-intelligent backtracking of Prolog based on a static data dependency analysis," in *Proc. Third Int. Logic Programming Conf.*, 1985.

[5] A. Ciepielewski and S. Haridi, "A formal model for OR-parallel execution of logic programs," in *Proc. Inform. Processing '83*, 1983, pp. 299–305.

[6] A. Ciepielewski and B. Hausman, "Performance evaluation of a storage model for OR-parallel execution of logic languages," in *Proc. 3rd IEEE Symp. Logic Programming*, Salt Lake City, UT, 1986, pp. 246–257.

[7] W. Citrin, "Parallel unification scheduling in Prolog," Ph.D. dissertation, Dec. 1986.

[8] W. F. Clocksin and C. F. Mellish, *Programming in Prolog*. New York: Springer-Verlag, 1981.

[9] J. S. Conery, "The AND/OR model for parallel interpretation of logic programs," Dep. Inform. Comput. Sci., Univ. of California, Irvine, 1983.

[10] J. Crammond, "A comparative study of unification algorithms for OR-parallel execution of logic languages," *IEEE Trans. Comput.*, vol. C-34, pp. 911–917, Oct. 1985.

[11] D. DeGroot, "Restricted AND-parallelism," in *Proc. Int. Conf. Fifth Generation Comput. Syst.*, 1984, pp. 471–478.

[12] P. Dembinski and J. Maluszynski, "AND-parallelism with intelligent backtracking for annotated logic programs," in *Proc. 1985 Symp. Logic Programming*, July 1985, pp. 29–38.

[13] J. B. Dennis and K. S. Weng, "An abstract implementation for concurrent computation with streams," in *Proc. 1979 Int. Conf. Parallel Processing*, Aug. 1979, pp. 35–45.

[14] T. Dobry, PLM Simulator Reference Manual, 1985.

[15] ——, "A high performance architecture for Prolog," Univ. of California, Berkeley, CA, Rep. UCB/Comput. Sci. Dept. 87/352, May 1987.

[16] C. Dwork, P. C. Kanellakis, and J. C. Mitchell, "On the sequential nature of unification," *J. Logic Programming 1*, June 1984, pp. 35–50.

[17] B. Fagin and T. Dobry, "The Berkeley PLM instruction set: An instruction set for Prolog," Comput. Sci. Division, Univ. of California, Berkeley, Sept. 1985. Res. Rep. UCB/Comput. Sci. Dept. 86/257.

[18] B. S. Fagin, "A parallel execution model for Prolog," Comput. Sci. Division, Univ. of California, Berkeley, CA, Rep. UCB/Computer Science Dept. 87/380, Nov. 1987.

[19] ——, "Performance studies of a parallel Prolog architecture," in *Proc. 14th Int. Symp. Comput. Architecture*, Pittsburgh, PA, June 1987, pp. 108–116.

[20] R. Hasegawa and M. Amamiya, "Parallel execution of logic programs based on dataflow concept," in *Proc. Int. Conf. Fifth Generation Comput. Syst.*, 1984, pp. 507–516.

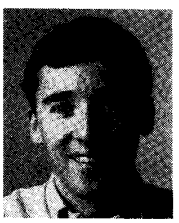
[21] P. Henderson, *Functional Programming*, 1980.

[22] M. V. Hermenegildo, "An abstract machine based execution model for computer architecture design and efficient implementation of logic programs in parallel," Dep. Comput. Sci., Univ. Texas at Austin, Austin, TX, Aug. 1986.

[23] K. Hwang, J. Ghosh, and R. Chowkwanyum, "Computer architectures for artificial intelligence processing," *IEEE Comput. Mag.*, vol. 20, pp. 19–27, Jan. 1987.

[24] P. P. Li and A. J. Martin, "The Sync model: A parallel execution method for logic programming," in *Proc. 3rd IEEE Symp. Logic Programming*, Salt Lake City, UT, 1986, pp. 223–234.

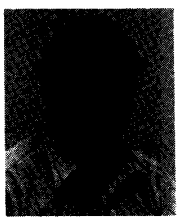
[25] G. Lindstrom, "OR-parallelism on applicative architectures," in *Proc. Second Int. Logic Programming Conf.*, July 1984, pp. 159-170.
[26] T. Moto-Oka, "The architecture of a parallel inference engine—PIE," in *Proc. Int. Conf. Fifth Generation Comput. Syst.*, 1984, pp. 479-488.
[27] L. Sterling and E. Shapiro, *The Art of Prolog*. Cambridge, MA: MIT Press, 1986.
[28] J. Syre and H. Westphal, "A review of parallel models for logic programming languages," ECRC Tech. Rep. CA-07, June 1985.



Barry S. Fagin (S'84-M'87) received the A.B. degree in engineering from Brown University, Providence, RI, in 1982, and the M.S. and Ph.D. degrees in computer science from the University of California, Berkeley, in 1984 and 1987.

While at Berkeley, he was a principal contributor to the Aquarius project, an investigation into high performance symbolic computing. Currently, he is an Assistant Professor of computer engineering at the Thayer School of Engineering, Dartmouth College, Hanover, NH. His research interests include computer architecture, special purpose computer design, and parallel symbolic computing.

Dr. Fagin is a member of the Association for Computing Machinery, and SIGARCH.



Alvin M. Despain (S'58-M'65) received the B.S., M.S., and Ph.D. degrees in 1960, 1964, and 1966, all from the University of Utah, Salt Lake City.

He has held faculty positions at Utah State University, Stanford University, and the University of California at Berkeley. His research interests include signal processing, multiprocessor architecture, computer architecture, and high performance computing; he is the author of over 60 papers in these fields. He has been a consultant to the electronics and computer industry for over 20 years. While at Berkeley, he was the principal investigator for the Aquarius research project. He is now a Professor of electrical engineering and computer science at the University of Southern California.

Dr. Despain is a member of Tau Beta Pi, Sigma Xi, Sigma Pi Sigma, Eta Kappa Nu, AAUP, ACM, and AAAI. He is a panel member of JASON, the SDIO Advisory Committee, and the ISAT working group for DARPA/ISTO.