PERFORMANCE STUDIES OF A PARALLEL PROLOG ARCHITECTURE

Barry S. Fagin
Alvin M. Despain

Computer Science Division
University of California
Berkeley, CA 94720

ABSTRACT

This paper presents a new multiprocessor architecture for the parallel execution of logic programs, developed as part of the Aquarius Project. This architecture is designed to support AND-parallelism, OR-parallelism, and intelligent backtracking. We present the most comprehensive experimental results available to date on combined AND-parallelism, OR-parallelism, and intelligent backtracking in Prolog programs. Simulation results indicate that most Prolog programs in use today cannot effectively make use of multiprocessing.

1. Introduction

The Aquarius project is a research effort concerned with the architectural issues of high performance computation. In particular, we are focusing on Prolog as the user language of the system. Previous work has led to the design of a high performance sequential Prolog architecture: the PLM [DDP85]. This paper presents the next phase of the research: an extension of the sequential architecture to a parallel one. First, we briefly discuss techniques for improving Prolog performance, and discuss a new execution model for Prolog: the PPP model (for Parallel Prolog Processor). We show how to extend the instruction set of the PLM to support the PPP, and present performance results of the architecture on a variety of benchmark programs. We conclude by analyzing the results of our experiments, and discuss future work.

Throughout this paper, the reader is assumed to be familiar with Prolog. For readers unfamiliar with Prolog, Clocksin and Mellish's text [ClM81] is an excellent introduction to the language.

2. Techniques for Improving Prolog Performance

Several techniques have been proposed for improving the performance of Prolog. Three of the most important are:

1) AND-parallelism
2) OR-parallelism
3) Intelligent backtracking

In this section, we briefly discuss each technique.

2.1. AND-parallelism

AND-parallelism was first suggested by Conery in his thesis [Con83]. AND-parallelism is the simultaneous solution of more than one goal in the body of a clause. For example, in the clause:

```
parents_of(Ch,M,F) :- mother(Ch,M), father(Ch,F).
```

the simultaneous solution of the "mother" and "father" goals would be an example of AND-parallelism.

The chief difficulty with AND-parallelism is the problem of binding conflicts. If no restrictions are placed on the goals to be solved in parallel, then it is possible that goals can bind shared variables to different values. For example, consider the clause:

```
tiger(X):- feline(X), carnivorous(X).
```

If the goals "feline" and "carnivorous" are executed in AND-parallel fashion, they might bind the variable X to different values. In addition to the synchronization overhead that this implies, a consistency check must also be performed, to filter out the bindings that do not match both goals.

This type of AND-parallelism, in which goals are allowed to bind shared variables, is called *unrestricted* AND-parallelism. Because this type of AND-parallelism entails considerable runtime overhead, most schemes for AND-parallelism incorporate compile-time analysis or program annotation to denote which goals produce and consume variable values [Con83], [DeM85]. These kind of schemes, in which AND-parallel goals execute only if guaranteed not to bind the same variables, utilize *restricted* AND-parallelism [DeG84]. Restricted AND-parallelism, as outlined by DeGroot, uses both compile-time and run-time checks to determine when goals can be executed in AND-parallel fashion without binding conflicts. The PPP employs a slightly narrower version of restricted AND-parallelism, using only compile time analysis to assist in the recovery of AND-parallelism [ChD85]. However, as in DeGroot's scheme, goals that run in AND-parallel fashion are guaranteed exclusive access to the variables they bind.

2.2. OR-parallelism

OR-parallelism was also identified by Conery as a potential source of concurrency in Prolog [Con83]. OR-parallelism is the simultaneous unification of multiple clauses in the program with the current goal. For example, in the program:

```
next_vertex(X,Y) :- connected(X,Y), ...

connected(1,2).
connected(1,3).
connected(2,5).
 ...
```

the simultaneous examination of all the clauses for "connected" by the goal "connected(X,Y)" would be an example of OR-parallelism.

Like AND-parallelism, the principal problem with OR-parallelism is that of multiple goals binding shared variables. Unlike AND-parallelism, however, compile-time analysis is of no value; OR-parallel goals by their very nature bind shared variables. Thus any scheme for supporting OR-parallelism must include a method for insulating the bindings generated by OR processes from one another.

A great deal of research has been devoted to solving this problem. The three main techniques were first proposed in [CiH83], [Bor84], and [Lin84]. The PPP uses the technique suggested in [Bor84], based on early indications of performance and the appropriateness of the scheme to our implementation of Prolog. For a more detailed explanation of OR-parallelism in the PPP, the reader is referred to [FaD87].

2.3. Intelligent Backtracking

When a Prolog goal fails, it 'backtracks' by resuming execution at the most recent point in the computation where alternative clauses existed to match the current goal. Such points are referred to as 'choice points'. However, the most recent choice point may not generate bindings relevant to the original cause of failure. For example, consider the program:

```
c(X,Y) :- a(X), b(Y).
```

and the query "c(X,3)?". If the goal "b(3)" has no solution, backtracking will generate another solution to "a(X)". But this will not affect the failure of b(3); further solutions to a(X) will be generated unnecessarily.

The ideal solution is to backtrack not to the most recent choice point, but to the most recent choice point that can produce bindings relevant to the cause of failure. This is called "intelligent backtracking", and was first suggested by Pereira et. al. [PeP80]. Substantial research has since been devoted to intelligent backtracking. Our work in this area builds on that of Chang and Despain [ChD85], [Cha85]. Chang introduces the concept of "semi-intelligent backtracking", in which backtracking occurs to the most recent choice point that can generate bindings appropriate to the cause of failure if such a choice point was placed on the stack by one of the goals in the current clause. In Chang's scheme, the intelligent backtracking possibilities are determined at compile time, and are not examined across clause boundaries; hence the term "semi-intelligent".

We have extended the work of Chang and Despain to support intelligent backtracking in a parallel execution environment. The necessary work is relatively simple; provisions must merely be made for killing and restarting parallel processes as a result of goal failure. This will be discussed in future work. For a more detailed discussion of semi-intelligent backtracking and parallel execution, the reader is referred to [Cha85].

3. The PPP Execution Model

Several execution models have been proposed for parallel logic programming, including the AND/OR process model [Con83], the "standard" OR-parallel model [CiH83], [Cra85], and the stack-based AND-parallel model [Bor84], [Her86]. The execution model of the PPP is discussed in [FaD87]. in which the PPP execution model is compared to other existing models. Here, we will briefly outline the execution model of the PPP to prepare the reader for the sections that follow.

3.1. Processes and Messages

Like other execution models for parallel Prolog, the PPP contains two kinds of processes: AND processes, and OR processes. An AND process corresponds in the obvious way to a goal in the body of a clause, while OR processes correspond to the clauses of a procedure. As their name suggests, AND processes execute in AND-parallel fashion and are the means by which AND-parallelism is exploited. Similarly, OR processes execute in OR-parallel fashion and are the means by which OR parallelism is utilized.

Processes communicate with each other through messages. There are four basics messages in the PPP: SUCCESS (SUC), FAIL, NEXT ANSWER (NA), and KILL. SUC is sent from a child to a parent to report the success of its original goal. FAIL is sent from child to parent to report failure. NA is sent from parent to child to induce backtracking, while KILL is sent from parent to child to terminate execution. Messages sent from child to parent (SUC and FAIL) are *informative*, while those sent from parent to child (NA and KILL) are *imperative*.

3.2. Process Behavior

Unlike other execution models, AND and OR processes in the PPP execute in an identical manner. The PPP differentiates between AND and OR processes when they send messages. The behavior of a process upon receiving a message is shown in table 1:

msg	condition	action
SUC	from OR	if parent ready, push bindings and continue; else put child to sleep
SUC	from AND	if siblings have finished continue; else record success and put child to sleep
FAIL	from OR	if parent waiting on this child, search for next child that has not failed. If none, backtrack. If next child has not yet succeeded, go to sleep, otherwise push bindings and continue. If parent not waiting on this child, record failure. Terminate child process.
FAIL	from AND	KILL all descendants of parent created after child, backtrack
NA		backtrack
KILL		kill descendants and terminate execution

Table 1

4. The Instruction Set Architecture of the PPP

Previous work on the Aquarius Project has resulted in the development of a high-performance architecture for Prolog: the PLM [DDP85]. Our work continues the work of Dobry by extending the PLM architecture to support the features of the PPP execution model. In this section we show how this extension is done.

The PLM is a special-purpose architecture for Prolog. Its instruction set is derived from the Prolog Instruction Set of Warren [War83]; a detailed description appears in [DPD84]. and [DDP85]. The new instructions necessary are those associated with AND-parallelism, OR-parallelism, and intelligent backtracking. This paper will discuss the instructions necessary for AND and OR parallelism. The new instructions necessary to incorporate intelligent backtracking into the PLM were introduced in [ChD85], and are not discussed here. The extensions necessary to extend Chang's intelligent backtracking algorithm to a parallel execution environment were relatively simple, and will appear in future work.

The original PLM instruction set appears in [DDP85]. The new instructions are shown in Table 2.

instruction	function	used for
i_allocate	begins code for clauses that use intelligent backtracking or AND-parallelism	AND-par, I.B.
call_p	creates AND process	AND-par
wait	synchronizes execution of parent with AND children	AND-par
try_p	creates first OR process in a procedure	OR-par
try_me_else_p	"	OR-par
retry_p	creates second through next-to-last OR processes in a procedure	OR-par
retry_me_else_p	"	OR-par
trust_p	creates last OR process in a procedure, puts process to sleep	OR-par
trust_me_else_p	"	OR-par

Table 2

"i_allocate" allocates a special environment on the runtime stack; it is used for clauses that can take advantage of AND-parallelism or intelligent backtracking.

"call_p" builds the data object on the stack necessary for indicating the presence of an AND process to the parent, and creates an AND-process for a particular goal.

"wait" is used to ensure that a process does not continue execution until all of a given group of AND-children have instantiated their variables. When the wait instruction is executed, the process examines a synchronization counter. If this counter is zero, it continues execution. If it is positive, the process goes to sleep. If other processes are available then this process may be swapped out.

"try_p" builds a data object on the stack used for controlling a group of OR processes, and creates an OR process for the first clause in a procedure. "retry_p" creates OR processes for the second through next-to-last clauses in a procedure. "trust_p" creates an OR process for the last clause in the procedure, and then puts the executing process to sleep.

"try_me_else_p", "retry_me_else_p", and "trust_me_else_p" are very similar to "try_p", "retry_p", and "trust_p". They differ only in where the OR processes they create begin execution and where execution continues in the current process. Since these instructions are included only as an aid to compilation, they are not essential to the completeness of the instruction set, and are not discussed further here.

5. Compiling Prolog for the PPP: An Example

We now show a small example of the compilation of a Prolog program. For the sake of illustration, we generate code that employs parallelism whenever possible, ignoring efficiency considerations. We will see that parallel code is only slightly larger than sequential code.

As an illustration of PPP compilation, we consider the following example [FaD87]:

```
main :- a(X), b(Y), c(X,Y).

a(1).
a(2).

b(1).
b(2).
b(3).

c(1,2).
```

5.1. Compilation for AND-parallelism

If the main clause of our example were compiled into sequential PLM code, the result would be:

```
procedure main/0
      allocate 2
      put_variable Y2,X1
      call a/1,2
      put_variable Y1,X1
      call b/1,2
      put_unsafe_value Y2,X1
      put_unsafe_value Y1,X2
      deallocate
      execute c/2
```

In the PPP, however, static analysis of the program reveals that the goals a(X) and b(Y) can be solved in parallel. Thus the compiler generates the following code:

```
procedure main/0
*     i_allocate 2,_BT,_JT,2
      put_variable Y2,X1
*     call_p a/1,1,1
      put_variable Y1,X1
*     call_p b/1,2,1
*     wait 1
      put_unsafe_value Y2,X1
      put_unsafe_value Y1,X2
      deallocate
      execute c/2
_BT:
_JT:
*     2
*     0
```

* changed from sequential code

Note that "allocate" has changed to "i_allocate", since the clause is now utilizing AND-parallelism. The call instructions have been changed to "call_p" instructions, and a wait instruction has been added to synchronize the parent with its children. A join table has been added to the clause at label _JT. Its only entry is 2, since two children must decrement it before the parent can succeed; the 0 is included only to indicate the end of the table to the assembler. Finally, a "backtrack table" [ChD85] could be inserted by the compiler at label "_BT", if intelligent backtracking were to be utilized.

5.2. Compiling for OR-parallelism

Next, consider the code for the 'a' procedure (the code for 'b' is similar). If compiled for sequential execution, the resulting code would be:

```
procedure  a/1
      switch_on_term  _614,fail,fail
        try _617
        retry _619
        trust _620
_617:
      get_constant  &1,X1
      proceed
_619:
      get_constant  &2,X1
      proceed
_620:
      get_constant  &3,X1
      proceed
_614:
# (Indexing instructions follow.
# These are used if the procedure is called
# with a constant argument)
          .
          .
          .
```

If we compile for OR-parallel execution, we obtain very similar code:

```
procedure a/1
      switch_on_term  _614,fail,fail
*       try_p 3,_617
*       retry_p 2,_619
*       trust_p 3,_620
_617:
      get_constant  &1,X1
      proceed
_619:
      get_constant  &2,X1
      proceed
_620:
      get_constant  &3,X1
      proceed
_614:
          .
          .
          .
```

* changed from sequential code

A try_p instruction replaces the try, which will create the data structure on the runtime stack needed to coordinate the transmission of bindings from a batch of OR-processes. It will also create an OR process to begin execution at label _617. The retry_p and trust_p instructions create the other two OR processes.

Note that the decisions to generate AND-parallel code for clauses and OR-parallel code for procedures can be made independently of one another; parallel clause code can be executed with sequential procedure code, and vice versa.

6. The Process Table and the PPP Process Kernel

While we have presented the instruction set of the PPP, many functions of the system remain to be discussed. For example, in order to achieve efficient processor utilization, we require a system-wide data structure that contains information about all processes in the system. In addition, we have not specified how the process behavior specified in table 1 is achieved. In this section we examine these two ideas more closely.

6.1. Processes in the PPP

A PPP process is a virtual PLM machine [DDP85]. This includes all registers in the PLM register set. Each process has its own writable address space, where its stack, heap, trail, and PDL are located[1]. A process can read from the address space of any of its ancestors, but with rare exceptions can write only to its own[2].

6.2. The Process Table

The PPP maintains a *process table* in main memory, which all processors can examine and update. Each entry in the process table is a PPP process. All processes currently in the system exist in the process table. In this way, the table can be examined for runnable processes when any processor becomes available.

6.3. The PPP Process Kernel

The actions denoted in table 1 are the responsibility of the PPP process kernel. The modules currently defined in the process kernel are shown in table 4.

We note that while the messages of the PPP model are a convenient abstraction, it is not necessary to actually send and receive messages in a PPP implementation. It is the action associated with each message that is important. Thus the process kernel does not deal with messages at all. Instead, it contains the routines that implement the semantic actions associated with the receipt of a given message by a given process. In effect, the modules of the process kernel serve to fill in the gaps in the implementation of the PPP left by the PLM instruction set; they are responsible for all actions of a process that affect other processes.

[1] For readers not familiar with these terms, these are the four memory areas of the PLM. A more detailed description of them is found in [DDP85].

[2] The exceptions occur when children update certain data structures in their parent's address space. For amore detailed description, see [FaD87].

PPP Process Kernel Modules	
name	function
PROCESS_SUCCESS()	take appropriate action when process succeeds
PROCESS_FAILURE()	take appropriate action when process fails
INDUCE_BACKTRACKING()	cause child to fail
PROCESS_CUT()	take appropriate action when process executes 'cut'
KILL_PROCESS()	kill indicated process
FORK()	create new process
DIE()	terminate process
NEWPROCESS()	scan process table for runnable processes, assign process to processor and continue
SLEEP()	put current process to sleep, call NEWPROCESS()

Table 4

7. The Aquarius Multiprocessor

One of the goals of the Aquarius Project is the construction of a high performance multiprocessor system that supports the PPP execution model. A diagram of this system is shown in figure 1:

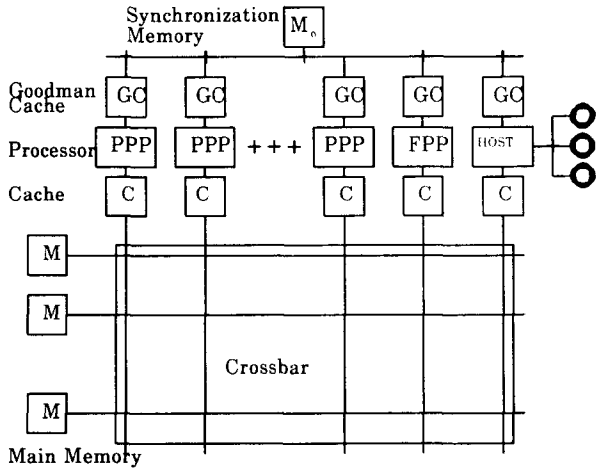

Figure 1: The Aquarius Multiprocessor

Each PPP is a separate Parallel Prolog Processor, suitably modified to execute the PPP instruction set. Each PPP is connected to two memory systems. The lower memory system consists of a cache for each processor, a crossbar, and sixteen memory modules. The upper system consists of a coherent cache and a synchronization memory, used to achieve atomic operations on shared data. Cache coherency algorithms and synchronization on the Aquarius Multiprocessor System are currently active areas of research [BiD86], but are not discussed in this paper.

The heap, stack, trail, and PDL for each process are part of the lower system, as is the process table. Each processor executes a PPP process, and can execute the routines of the process kernel to create new processes, take the appropriate action upon success and failure, and so forth.

8. The Simulator

In this section we discuss the simulator used to obtain our performance results.

The PPP simulator is an extension of the PLM simulator of Dobry [Dob85] In addition to modeling all sequential PLM instructions, the PPP simulator includes the new instructions of the PPP and the routines of the process kernel.

In attempting to measure PPP performance, we were faced with the problem of evaluating a system containing components in varying stages of development. The processors themselves are based on the PLM, so the execution times of sequential PLM instructions are known from [DDP85]. In addition, the new instructions of the PPP can be accurately estimated, based on a knowledge of their actions they perform and the PLM datapath. There are, however, two sources of uncertainty in modeling the PPP: the memory system architecture, and the operating system.

8.1. Approximations in the PPP Simulator

Currently, the memory system architecture of the Aquarius multiprocessor is under development, so we have not attempted to model it. All reads and writes are assumed to execute in one cycle; contention for memory modules, the degradation of cache performance due to context swapping, and other features of memory system behavior are not modeled. However, since such modeling would reduce the performance estimates even further, this strengthens our conclusions regarding the inability of most Prolog benchmark to effectively exploit multiprocessing.

In addition, the operating system software for Aquarius currently exists only as a collection of C routines in the PPP simulator. Thus timing estimates for system calls are difficult to make. To address this problem, we identify certain operations as fundamental, supplying their associated execution times as parameters to the simulation. The simulator can then use these values to estimate performance.

The operations we identified as fundamental and the values used in our simulation runs are shown below:

operation	#cycles
process creation	50
process termination	25
invoke scheduler	50
interrupt running process	50
sync mem access	5

Table 5

Process creation in the PPP consists of the initialization of a virtual PLM machine, the assignment of a portion of the address space, and the allocation of a slot in the process table. Process termination involves releasing the memory of a process and freeing up its slot in the table. An invocation of the scheduler consists of finding a runnable process and loading it on an available processor. Processors may interrupt one another on instruction boundaries; when this occurs it is assumed to take 50 clock cycles. Finally, an access to the synchronization memory is assumed to take 5 clock cycles.

We note that our assumptions are quite idealistic. However, this strengthens our conclusion that most Prolog programs being written today cannot effectively utilize concurrency, owing to the costs involved. We shall see evidence of this shortly.

8.2. Timing Estimation

The PPP simulator is an event-driven multiprocessor simulator. An arbitrary number of processors (16 by default) may be simulated. The simulation is interleaved at the PPP instruction level: one processor is simulated for one PPP instruction, then the next, and so forth.

The simulator maintains a clock for each processor. Each time a routine is called, the clock of the processor being simulated is incremented an appropriate number of cycles. The simulator also keeps track of the number of fundamental operations executed on each simulation step, and increments the clock appropriately each time an operation is executed.

Frequently, processors will be come idle at the same time as others become busy. When a processor becomes idle, this is recorded in the simulator. When the processor becomes busy again, its clock is set equal to the clock of the processor responsible for waking it up. This is done to ensure correct timing estimates for processors that become idle. Suppose, for example, that at the beginning of a simulation run processor 0 were to run for 1000 cycles and then create a process to run on previously idle processor 1. Processor 1's clock would then begin at 1000. Suppose further that processor 0 then becomes idle, while processor 1 runs for 500 cycles before performing an action that causes processor 0 to wake up. Upon waking up, processor 0's clock would be set to 1500, (and not left at its original value of 1000), to reflect the time dependency between events of the simulation.

9. Performance Results

We now present the performance estimates of the PPP simulator on a variety of benchmark programs. These results are the most comprehensive performance analysis of Prolog programs known to the authors. They consider a wide variety of Prolog programs, compiled for AND-parallelism, OR-parallelism, and intelligent backtracking. They also take into account process creation time, scheduling overhead, and other factors crucial to any parallel logic programming implementation.

Our initial performance estimates of a four processor system are shown in table 6.

These benchmarks were all taken from [War80], [ClM81], and the set of programs distributed over the arpanet, with the exception of cckt4 which was written at Berkeley.

Some programs that appear to be able to exploit AND- or OR-parallelism do not have an entry in this table. This is because occasionally a program will create more processes than the simulator can manage. This is the case, for example, with the query benchmark and OR-parallelism.

In addition, some entries of the table contain values greater than 4, even though only four processors are being simulated. This is due to the effects of intelligent backtracking, which enhance the performance of Prolog even if only one processor is utilized.

Finally, it should be noted that with the exception of con6, the nondeterministic concatenate benchmark, all our programs compiled for OR-parallelism were written to compute a single answer to a query, and not all possible solutions. In a program where all solutions are desired, OR-parallelism is equivalent to AND-parallelism in the sense that all processes perform useful work. This in turn leads to larger speedups. However, we feel that such a use of OR-parallelism is unrealistic; the vast majority of the time the user is interested in one answer that satisfies a set of constraints, and not all possible solutions. Thus we have simulated benchmarks that compute a single answer, and not an entire set.

Expected Speedup of Benchmarks
(4 processors)

prog	A	O	I	AO	AI	OI	AOI
divide10	0.80	x	x	x	x	x	x
log10	x	x	x	x	x	x	x
ops8	1.13	x	x	x	x	x	x
times10	0.75	x	x	x	x	x	x
palin25	1.07	0.10	x	x	x	x	x
qsd	1.57	0.08	x	0.10	x	x	x
query	0.95	x	x	0.47	x	x	x
con1	x	x	x	x	x	x	x
con6	x	0.37	x	x	x	x	x
hanoi	x	x	x	x	x	x	x
ckt4	1.07	1.22	7.57	0.85	10.14	6.68	5.27
mumath	x	2.13	x	x	x	x	x
queens	0.69	2.12	x	2.01	x	x	x
deep_bak	x	0.29	x	x	x	x	x
envir	0.32	x	x	x	x	x	x
map	x	0.11	x	x	x	x	x
knight	x	0.39	x	x	x	x	x

Table 6[3]

10. Analysis of Results

We see that even under very idealistic assumptions, most programs exhibit little speedup. In fact, many run slower despite their theoretical potential concurrency. This is due to the creation of processes where the costs of concurrent computation outweigh the benefits. This indicates an important point:

The visual inspection of a Prolog program is not enough to detect useful concurrency.

The decision to execute a piece of code with a separate process cannot be made by simply looking at the program. It must be made according to a careful analysis of the costs and benefits associated with process creation. We are currently constructing a cost/benefit model to assist in this area.

[3] In this table, the letters A, O, and I refer to compilation for AND-parallelism, OR-parallelism, and intelligent backtracking respectively. The letter 'x' indicates that the given benchmark could not make use of a particular technique.

10.1. Doubling the Number of Processors

If the number of processors is increased to eight, the estimated performance is shown in table 7.

Expected Speedup of Benchmarks
(8 processors)

prog	A	O	I	AO	AI	OI	AOI
divide10	1.03	x	x	x	x	x	x
log10	x	x	x	x	x	x	x
ops8	1.45	x	x	x	x	x	x
times10	1.03	x	x	x	x	x	x
palin25	1.07	0.10	x	x	x	x	x
qsd	1.68	0.08	x	0.13	x	x	x
query	0.95	x	x	0.71	x	x	x
con1	x	x	x	x	x	x	x
con6	x	0.49	x	x	x	x	x
hanoi	x	x	x	x	x	x	x
ckt4	1.07	1.45	7.57	1.28	10.14	10.25	10.05
queens	0.73	3.89	x	2.07	x	x	x
deep_bak	x	0.29	x	x	x	x	x
envir	0.56	x	x	x	x	x	x
map	x	0.11	x	x	x	x	x
knight	x	0.76	x	x	x	x	x

Table 7

We see that in most cases the speedup does not increase significantly; only the ckt4 benchmark shows appreciable gains. This suggests two possibilities: either the system is so overwhelmingly saturated with processes that doubling the number of processors from four to eight has little effect, or the average number of runnable processes is relatively low. The results of the next section indicate the latter.

10.2. Breakdowns of Execution Time

Space prevents a complete breakdown of execution time for all the combinations of AND-parallelism, OR-parallelism, and intelligent backtracking. Results for AND- and OR-parallelism for four and eight processor simulation runs are shown below:

Proportion of Total Time: AND
(4 processors)

	AND							
prog	UIN	SYS	SYN	PCR	PTR	SCH	IN	idle
divide10	24.7	8.6	1.9	3.2	0.0	9.4	0.0	52.2
ops8	34.5	8.8	2.6	4.3	0.0	11.1	0.0	38.7
times10	23.9	9.6	2.2	3.6	0.0	10.4	0.0	50.5
palin25	28.8	1.0	0.3	0.5	0.0	2.7	0.0	66.6
qsd	42.5	6.6	1.5	2.5	0.0	8.2	0.0	38.7
query	46.7	0.5	1.8	0.0	0.0	3.3	0.0	47.7
ckt4	35.2	0.9	0.4	0.7	0.3	2.5	0.6	59.3
queens	25.2	4.0	1.9	2.7	1.4	8.0	2.0	54.9
envir	22.0	29.7	5.9	9.9	0.0	28.4	0.0	4.1
avg	31.5	7.8	2.1	3.0	0.2	9.3	0.3	45.9

Table 8

Proportion of Total Time: OR
(4 processors)

	OR							
prog	UIN	SYS	SYN	PCR	PTR	SCH	IN	idle
palin25	58.5	2.3	0.9	1.5	0.5	3.1	0.1	33.1
qsd	54.6	3.0	1.0	1.9	0.5	3.4	0.4	35.2
con6	35.0	10.1	4.0	5.7	2.8	14.2	0.0	28.1
ckt4	70.4	1.4	0.0	0.1	0.0	2.5	0.0	25.6
mumath	63.1	8.6	2.7	4.2	2.0	11.3	0.0	8.0
queens	70.7	7.9	2.8	4.5	2.0	10.8	0.0	1.3
deep_bak	9.8	12.8	6.1	10.3	4.9	20.6	0.0	35.4
map	46.2	10.2	4.9	10.6	2.7	18.8	0.0	6.6
knight	80.4	6.9	1.4	2.2	0.9	8.1	0.0	0.1
avg	54.3	7.0	2.7	4.5	1.8	10.3	0.1	19.3

Table 9

Proportion of Total Time: AND
(8 processors)

	AND							
prog	UIN	SYS	SYN	PCR	PTR	SCH	IN	idle
divide10	15.5	4.1	1.2	2.0	0.0	5.9	0.0	71.3
ops8	30.3	6.2	2.3	3.8	0.0	9.8	0.0	47.7
times10	16.1	4.8	1.5	2.4	0.0	7.0	0.0	68.2
palin25	15.2	0.5	0.2	0.3	0.0	1.4	0.0	82.5
qsd	26.8	3.7	1.0	1.6	0.0	4.9	0.0	62.0
query	46.7	0.5	1.8	0.0	0.0	3.3	0.0	47.7
ckt4	35.2	0.9	0.4	0.7	0.3	2.5	0.6	59.3
queens	14.6	2.2	1.1	1.5	0.8	4.5	1.1	74.2
envir	21.7	28.6	5.9	9.8	0.0	27.6	0.0	6.4
avg	24.7	5.7	1.7	2.4	0.1	7.4	0.2	57.7

Table 10

Proportion of Total Time: OR
(8 processors)

	OR							
prog	UIN	SYS	SYN	PCR	PTR	SCH	IN	idle
palin25	37.3	1.4	0.6	1.0	0.3	2.0	0.0	57.4
qsd	54.9	3.0	1.0	1.9	0.5	3.4	0.4	35.0
con6	19.6	4.6	2.2	3.2	1.6	8.0	0.0	60.8
ckt4	59.4	0.5	0.0	0.1	0.0	2.1	0.0	37.8
mumath	53.0	7.1	2.3	3.5	1.7	9.5	0.0	22.9
queens	66.1	7.3	2.6	4.4	1.9	10.3	0.0	7.3
deep_bak	6.8	8.8	4.2	7.1	3.4	14.3	0.0	55.4
map	46.2	10.2	4.9	10.6	2.7	18.8	0.0	6.6
knight	80.1	6.8	1.4	2.1	0.9	8.1	0.0	0.5
avg	47.0	5.5	2.1	3.8	1.4	8.5	0.1	31.5

Table 11

We see that for both types of parallelism, increasing the number of processors increased the percentage of time that processors spent idle. This indicates that the benchmarks we have examined are inherently sequential: they have difficulty keeping even a small number of processors busy.

10.3. The Effect of Increased Process Creation Time

One of the advantages of the PPP simulator is that the effect of changing the cost of one fundamental operation can be examined while others are held constant. For example, if process creation time is assumed to take 1000 cycles (a much more realistic value than the 50 initially assumed), then the estimated performance of a four processor system is as follows:

Expected Speedup of Benchmarks
(4 processors)

under the following assumptions:

UIN	SYS	SYN	PCR	PTR	SCH	IN
1	1	5	1000	25	50	50

prog	A	O	I	AO	AI	OI	AOI
divide10	0.43	x	x	x	x	x	x
log10	x	x	x	x	x	x	x
ops8	0.52	x	x	x	x	x	x
times10	0.38	x	x	x	x	x	x
palin25	0.94	0.07	x	x	x	x	x
qsd	0.98	0.06	x	0.07	x	x	x
query	0.95	x	x	0.20	x	x	x
con1	x	x	x	x	x	x	x
con6	x	0.13	x	x	x	x	x
hanoi	x	x	x	x	x	x	x
ckt4	0.73	1.21	7.57	0.66	6.75	6.45	3.86
mumath	x	1.08	x	x	x	x	x
queens	0.23	1.08	x	0.67	x	x	x
deep_bak	x	0.04	x	x	x	x	x
envir	0.11	x	x	x	x	x	x
map	x	0.11	x	x	x	x	x
knight	x	0.28	x	x	x	x	x

Table 12

We see that as expected, performance is reduced, although not as significantly as one might expect. The increase in process creation time has a greater effect on OR-parallel performance than AND-parallel performance. This agrees with the data of the previous section, which indicates that programs in the benchmark set that exploited OR-parallelism spent a slightly higher portion of their execution time creating processes than AND-parallel programs.

Space prevents further analysis of this and other simulation data. A more extensive study of parallel Prolog performance will appear in [Fag87].

11. Conclusions

We have presented the instruction set and architecture of the Aquarius Multiprocessor System, based on the PPP execution model outlined in [FaD87]. We have also presented the first detailed simulation results of Prolog programs compiled for AND-parallelism, OR-parallelism, and intelligent backtracking.

Our results indicate that most Prolog benchmarks being written today are not likely to be sped up by parallel processing, due to either their inherent sequentiality, the costs associated with multiprocessing, or both. In order to effectively utilize the processing power of multiprocessors, parallel Prolog programs must create processes only when benefits are believed to outweigh costs. For systems with high process creation overhead, this would imply that relatively few processes should be created, with each process doing a great deal of work.

It should be noted that our work tests only the amount of concurrency recoverable using the PPP execution model. There may be other execution models that can extract more concurrency, but at this time no other simulation studies of this type of execution model are available. In addition, the PPP is theoretically more powerful than most parallel execution models for Prolog, supporting restricted AND-parallelism, OR-parallelism, and intelligent backtracking. Thus it is capable of extracting major sources of performance improvement in Prolog. Since a great deal of effort in the past few years has been spent on parallel execution models for logic languages, we believe it will be very difficult to develop realistic execution models that can extract more concurrency than the PPP.

We would caution, however, against making strong generalizations regarding the efficacy of a parallel Prolog on the basis of our results. Benchmarks are not natural phenomena; they are artifacts of intelligence, constructed for a particular purpose to run on a particular machine. We face a chicken and egg problem in that parallel Prolog programs that can solve useful problems using multiprocessing are not likely to be written until the tools exist to develop and evaluate them. The existing tools for sequential Prolog program development (interpreters, compilers, simulators, and sequential processors) are simply inadequate for the task. Thus the final verdict on the utility of parallel Prolog systems must wait until explicitly parallel benchmarks are developed and analyzed on existing multiprocessor hardware.

12. Future Work

The results presented here are very much work in progress. Research is continuing in a number of areas. We are currently investigating the issues involved in implementing the process kernel efficiently. We are also developing an intelligent compiler for the PPP that can make good decisions concerning the generation of parallel versus sequential code.

In addition, we are studying better ways of managing memory in the PPP. Allocating separate writable address spaces for each process in the PPP, while encouraging efficient processor utilization and permitting the utilization of OR-parallelism [FaD87], also increases the complexity of memory management. Assigning one address space per processor, as Hermenegildo [Her86] and Borgwardt [Bor84] suggest, drastically simplifies many aspects of the system. It may be possible to incorporate the desirable memory management properties of this model into the PPP, to simplify memory management while maintaining support for OR-parallelism and efficient processor utilization.

13. Acknowledgements

Work on the PPP execution model was begun by Tep Dobry, who was also the principal architect of the PLM. The authors are especially appreciative of his contributions. We thank the referees for their careful review of the paper and constructive comments. We are grateful for the interaction with the other principal investigators of the Aquarius effort: Professors Yale N. Patt and Vason P. Srini. Finally, we have benefited considerably from discussions with the other members of Aquarius: Phil Bitar, Bill Bush, Mike Carlton, Chien Chen, Wayne Citrin, Ron David, Jeff Gee, Bruce Holmer, Wen-Mei Hwu, Rick McGeer, Tam Nguyen, Ashok Singhal, Jerric Tam, Jim Testa, Herve' Touati, and Robert Yung.

This work was sponsored by the Defense Advanced Research Projects Agency, Arpa Order No. 4871, and is monitored by Space and Naval Warfare Systems Command under Contract No. N00039-84-C-0089.

14. References

[BiD86] P. Bitar and A. Despain, "Multiprocessor Cache Synchronization", *Proceedings of the 13th International Symposium on Computer Architecture*, Tokyo, Japan, Jun. 1986, 424-433.

[Bor84] P. Borgwardt, "Parallel Prolog Using Stack Segments on Shared-Memory Multiprocessors", *Proceedings of the 1984 Symposium on Logic Programming*, Feb. 1984, 2-11.

[Cha85] J. H. Chang, "High Performance Execution of Prolog Programs Based on a Static Data Dependency Analysis", *Ph. D. Thesis*, Berkeley, Oct. 1985. UCB/Computer Science Dpt. Research Report No. 86/283.

[ChD85] J. H. Chang and A. M. Despain, "Semi-Intelligent Backtracking of Prolog Based on A Static Data Dependency Analysis", *Proceedings of the Third International Logic Programming Conference*, 1985.

[CiH83] A. Ciepielewski and S. Haridi, "A Formal Model For OR-Parallel Execution of Logic Programs", *Proceedings of Information Processing '83*, 1983, 299-305.

[ClM81] W. F. Clocksin and C. F. Mellish, *Programming in Prolog*, Springer-Verlag, New York, NY, 1981.

[Con83] J. S. Conery, *The AND/OR Model for Parallel Interpretation of Logic Programs*, Dept. of Information and Computer Science, University of California, Irvine, 1983.

[Cra85] J. Crammond, "A Comparative Study of Unification Algorithms for OR-Parallel Execution of Logic Languages", *IEEE Transactions on Computers C-34* (Oct. 1985), 911-917, IEEE.

[DeG84] D. DeGroot, "Restricted AND-Parallelism", *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1984, 471-478.

[DeM85] P. Dembinski and J. Maluszynski, "AND-Parallelism with Intelligent Backtracking for Annotated Logic Programs", *Proceedings of the 1985 Symposium on Logic Programming*, Jul. 1985, 29-38.

[DPD84] T. P. Dobry, Y. N. Patt and A. M. Despain, "Design Decisions Influencing the Microarchitecture for a Prolog Machine", *Proceedings of the 17th International Symposium on Microarchitecture*, Oct. 1984.

[Dob85] T. Dobry, PLM Simulator Reference Manual, 1985.

[DDP85] T. P. Dobry, A. M. Despain and Y. N. Patt, "Performance Studies of a Prolog Machine Architecture", *Proceedings of the 12th Annual International Symposium on Computer Architecture*, Jun. 1985, 180-190.

[FPD85] B. Fagin, Y. Patt, A. Despain and V. Srini, "Compiling Prolog Into Microcode: A Case Study Using the NCR-32000", *Proceedings of the 18th International Symposium on Microarchitecture*, 1985, 79-88.

[FaD87] B. Fagin and A. Despain, Combining AND and OR Parallelism in Logic Programs, (to appear), May 1987.

[Fag87] B. Fagin, *A Parallel Execution Model for Prolog*, University of California, Berkeley, CA, Jun 1987. Ph.D. Thesis (in progress).

[Her86] M. Hermenegildo, "Efficient Management of Backtracking in AND-Parallelism", *Proceedings of the 3rd International Conference on Logic Programming*, London, 1986.

[Lin84] G. Lindstrom, "OR-Parallelism on Applicative Architectures", *Proceedings of the Second International Logic Programming Conference*, Jul. 1984, 159-170.

[PeP80] L. M. Pereira and A. Porto, *An Interpreter of Logic Programs Using Selective Backtracking*, Universade Nova de Lisboa, Jul. 1980. Report 3/80, Dept. de Informatica.

[War80] D. H. D. Warren, "Logic Programming and Compiler Writing", *Software -- Practice and Experience 10* (Feb. 1980), 97-126.

[War83] D. H. D. Warren, *An Abstract Prolog Instruction Set*, Computer Science and Technology Division, SRI, Menlo Park, CA, Oct. 1983. Technical Note 309, Artificial Intelligence Center.