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

A Parallel Execution of Functional Logic Language with Lazy Evaluation

Jong H. Nang, D. W. Shin, S. R. Maeng, and Jung W. Cho
Department of Computer Science & Center for Artificial Intelligence Research
Korea Advanced Institute of Science and Technology
373-1 Kusong-Dong, Yusung-Ku, Taejon 305-701, Korea
e-mail : jhnang@adam.kaist.ac.kr

Abstract

In this paper, we propose a parallel computational model, called PR^3 (Parallel Resolution and Reduction with RAP), and its abstract machine for the parallel execution of Lazy Aflog program. Lazy Aflog together with its abstract machine FWAM-II was proposed as a cost-effective functional logic language. Since the parallel reduction of the arguments of a function can be regarded as a parallel evaluation of independent subgoals, only Independent-And Parallelism is exploited in PR^3 in order to simplify the execution control. PR^3 is an extension of DeGroot's RAP, and it is proposed as a simple and coherent parallelizing method that can be applied both of the function and logic. A parallel abstract machine for PR^3 based on the RAP-WAM is also developed, which is an extension of FWAM-II equipped with the run-time structures and the primitive instructions to spawn the parallel executions and gather the results. Simulation results show that both of the parallel resolution and parallel lazy reduction can be provided efficiently in the PR^3 and abstract machine.

1 Introduction

During the last couple of decades, there has been growing interest in functional languages and logic languages as potential alternatives to conventional languages, because of their declarative semantics and no side-effect. They have been widely used as system programming as well as application programming languages. Functional languages are characterized by reduction rules which make them procedural, while logic languages have the declarative flavour owing to their logical backgrounds. However, there exist software components which include both procedural and declarative part. Since defining them in one paradigm, procedurally or declaratively, would be unnatural and leads to inefficiency [Bellia and Levi 1986], there have emerged a lot of research efforts on the combination of two languages.

Lazy Aflog [Nang *et al.* 1991] is an E-Unification (Equality-Unification) based functional logic language, in which an E-Unification, called *E-Unification with lazy evaluation*, is developed to combine the lazy reduction of functional language and two-way argument passing of logic languages. Thanks to this E-Unification, the noticeable functional language features such as infinite data structures and higher-order function can be expressed naturally, while the expressiveness of the logic language such as non-determinism and unification is also maintained in the single framework. FWAM-II [Nang *et al.* 1991] is an abstract machine for Lazy Aflog, in which instructions and run-time structures to support the suspension and reactivation of functional closure are incorporated into WAM. We already demonstrated in [Nang *et al.* 1991] that this pair would be a good compromise between the expressiveness and efficiency of the combination.

Although FWAM-II is designed to maximize the performance on the conventional von-Neumann computers, it has the speed limitation because of its sequential nature. A natural way to improve the performance is to extend Lazy Aflog and FWAM-II pair in parallel, while keeping the performance optimizations and storage efficiency of sequential system. However, parallelizing Lazy Aflog computation is not a trivial problem, because we should deal with two different styles of parallelisms, one for logic part and the other for functional part. The simplest way in parallelization is to adopt already developed parallelizing schemes for each part, for example, Conery model [Conery 1983] for logic part and parallel graph reduction model such as GRIP [Peyton Jones *et al.* 1987] for functional part. It, however, requires a complex control mechanism to switch between the parallel execution of logic and functional part. Hence, instead of having two different schemes, it is highly desirable to develop a coherent one that could be applied to both logic and functional part.

Since the main parallelism in the functional part of Lazy Aflog program is the parallel reduction of arguments and it can be viewed as an Independent-AND Parallelism in the view point of logic language, the parallelisms in both parts can be exploited easily if there is a parallelizing method for Independent-AND Parallelism. The RAP Model [DeGroot 1984] is such a parallelizing method to spawn the parallel executions when there are independent subgoals in a clause. In this paper, we propose a parallel execution model for Lazy Aflog, called PR^3 (Parallel Resolution and Reduction with RAP), which is an extension of RAP. In PR^3, only independent subgoals in a clause and all the arguments of a strict function are resolved and reduced in parallel. Although this approach overlooks some available parallelisms in a Lazy Aflog program such as OR-Parallelism in logic part, it helps to avoid the complex run-time support.

In addition, this paper proposes an abstract machine for PR^3, called PFWAM-II (Parallel FWAM-II). It is an extension of FWAM-II equipped with the run-time structures and primitive instructions to spawn the parallel execution and gather the results. These run-time structures and instructions are inherited from the RAP-WAM [Hermenegildo 1986] with some modifications for the parallel lazy reduction of functional terms. Simulation

results to show the p-processor speed-up ratio over single processor are also presented to show the efficiency of PR^3 and PFWAM-II.

This paper is structured as follows. Section 2 briefly recalls our previous works on the Lazy Aflog and FWAM-II. A parallel computational model based on RAP for Lazy Aflog is presented in Section 3, while a parallel extension of FWAM-II for the parallel model is followed in Section 4. The simulation results that show the performance of the parallel extensions and a comparison with the related works are presented in Section 5. Finally, a summary of paper is presented in Section 6.

2 Lazy Aflog and FWAM-II

Lazy Aflog [Nang *et al.* 1991] is a successor of Aflog [Shin *et. al* 1987, Shin *et al.* 1988], to which the capability to process infinite data structures and higher-order function are added. FWAM-II is also a successor of the abstract machine for Aflog [Shin *et al.* 1992] with the primitives to suspend and reactive the functional closure at the machine instruction level. Lazy Aflog and FWAM-II pair was proposed as an effective mechanism to incorporate the functional features into logic. Now, let us explain Lazy Aflog and FWAM-II in more detail.

A Lazy Aflog program consists of a set of Prolog clauses and a set of function definitions (or rewrite rules) written in a constructor based functional language. The functional symbols in Lazy Aflog programs are classified into two disjoint sets : a set of *constructors* and a set of *defined functions*. A symbol f is a defined function if it appears at the left hand side of a rewrite rule, otherwise it is treated as a constructor symbol. In a Lazy Aflog program, a function application occurs as an argument of Prolog subgoal, which is reduced to its WHNF (Weak Head Normal Form) [Peyton Jones 1986] when the subgoal is resolved. This is the way in Lazy Aflog to incorporate functional programming into logic programming. Lazy Aflog imposes a restriction that all the arguments of a function should be ground before the function is reduced. Even though it prevents Lazy Aflog from having the powerful inferencing mechanism such as narrowing, it greatly contributes to the efficiency of the underlying E-Unification algorithm, because it assures that the E-unifier of two terms is unitary.

Let us explain the programming style and operational semantics of Lazy Aflog. Example 1 is the famous *Sieve of Eratosthenes* program which generates the list of all the prime numbers infinitely using lazy evaluation technique.

Example 1 *Sieve-of-Eratosthenes*

```
C1 : test(X) :- truncate(X, sieve(from(2))).
C2 : truncate(0,L).
C3 : truncate(X,[H | T]) :-
        print_era(X,H), Y is X - 1, truncate(Y,T).
C4 : print_era(X,H) :- write(X), tab(2), write(H), nl.

F1 : from(N) ==> [N | from(N+1)].
F2 : sieve([P | L]) ==> [P | sieve(filterp(P,L))].
F3 : filterp(P, [X | L]) ==> ((X%P) == 0 | filterp(P,L))
F4 :                         [X | filterp(P,L)].
```

In Example 1, a query ":- test(100)." generates 100 consecutive prime numbers as its result. In the course of the refutation of the query, the unification of *truncate(100, sieve(from(2))* in C_1 and *truncate(X, [H | T])* in C_3 is tried as follows :

```
        call E-Unify(sieve(from(2)), [H | T])
→         call E-Unify(from(2), [P | L])   /* by F2 */
→         exit E-Unify([2 | from(2+1)],
                       [2 | from((2+1))])   /* by F1 */
→       call E-Unify(sieve([2 | from(2+1)], [H | T])
→       exit E-Unify([2 | sieve(filterp(2, from(2+1)))],
                     [2 | sieve(filterp(2,from(2+1)))])   /* by F2 */
```

In this E-Unification† process, the reduction of a functional term is initiated when a head pattern of a clause or rewrite rule is a non-variable term and the corresponding argument of the caller is a functional term. Note that the functional term is not completely reduced to its normal form, but to WHNF, which makes it possible to handle the infinite data structures. The complete description of the E-Unification algorithm, called *E-Unification with Lazy Evaluation*, is presented in [Nang *et al.* 1991].

FWAM-II, an abstract machine for Lazy Aflog, is an extension of WAM augmented with the manipulation of functional closure. It is characterized by that :

- it adds the reduction mechanism to the WAM architecture, and
- it employs an environment-based reduction rather than graph reduction.

Since WAM uses an environment for the variables in the body of a clause, the conventional environment-based reduction scheme is more suitable to WAM than the graph reduction is in the combination. Therefore, FWAM-II behaves similarly to the WAM in the execution of a clause, whereas it works similarly to an environment-based reduction machine in the reduction of functional term. This WAM-based approach has been also adopted in other abstract machines for the functional logic language, such as K-WAM [Bosco *et al.* 1989] for K-LEAF and a WAM model [Nadathur and Jayaraman 1989] for λ-Prolog. The E-unification of Lazy Aflog is realized in FWAM-II via the reducibility checking in the unification instructions, which immediately calls the reduction process if the passed argument is a functional term and corresponding pattern is not a non-variable term. To implement the suspension and reactivation of functional closure, a run-time structure (called, *Reduction Stack*) is added to WAM structure. Figure 1 shows a compiled FWAM-II code for the *filterp* function in Example 1, where *mode* and *eq* are predefined strict functions.

Upon the benchmark testing [Nang *et al.* 1991], the reduction mechanism of FWAM-II is relatively less efficient than WAM executing pure Prolog programs because of its overhead to construct and reference the functional closure, but it can support lazy evaluation in logic in the abstract machine level. Consequently, it is argued that FWAM-II can support not only all the features of logic language but also the essential features of functional language with the performance comparable to WAM.

3 A Parallel Computational Model for Lazy Aflog

Although FWAM-II would be an efficient sequential abstract machine for Lazy Aflog, it has the speed limitation because of its sequential nature. A natural way to overcome this obstacle is to extend it in parallel. This

```
F1 : filterp(P, [X | L]) => ((X%P) == 0 | filterp(P,L))
F2 :                        [X | filterp(P,L)].
```

```
F1:   allocate                3
      % Pattern Matching
      fget_value              'P', X1
      fget_list               X2
      match_value             'X'
      match_value             'L'
      %Guard Checking
      try_me_else_L           F2
      put_value               X, X1
      put_value               P, X2
      call_P_Arity_N          mode/2, 2
      put_integer             0, X2
      call_P_Arity_N          eq/2, 2
      %Committing
      commit
      %Construct WHNF
      write_function          'filterp/2', X1
      write_value             'P'
      write_value             'L'
      rewrite_value           X1
      % Returning
      return
F2:   trust_me_else_fail
      write_function          'filterp/2', X1
      write_value             'P'
      write_value             'L'
      write_list              X2
      write_value             'X'
      write_value             X1
      rewrite_value           X2
      return
```

Figure 1 A Compilation Example

section addresses our point of view that adopts the RAP as our starting point, and presents a parallel computational model for Lazy Aflog.

3.1 Parallelisms in Lazy Aflog Programs

Lazy Aflog has various kinds of parallelisms inherited from both function and logic, such as AND-Parallelism, OR-Parallelism, and Argument-Parallelism. Among these parallelisms, we adopt the Independent AND-Parallelism as the primary parallelism owing that :

- Ideally, all parallelisms in the Lazy Aflog program can be exploited in the parallel extension. However, it may require a complex control mechanism that may degrade the performance gains obtained through the parallel execution.
- Since the Argument-Parallelism in the functional language part can be viewed as a kind of Independent-AND Parallelism in the logic language part, we can exploit parallelisms in both of the functional and the logic parts in a simple and coherent manner if there is a parallelizing method for it.
- There have emerged an efficient and powerful computational model and an abstract machine for Independent-AND Parallelism of logic programs. DeGroot's RAP Model and RAP-WAM [Hermenegildo 1986] are such a computational model and an abstract machine, respectively.

3.2 A Parallel Computational Model : PR^3

A parallel computational model for Lazy Aflog, called PR^3 [Nang 1992], is a parallel model which can support both of the parallel resolution and parallel lazy reduction simultaneously. The basic principle to spawn a parallel task is as follows ;

Rule 1) the subgoals in a clause are executed in parallel when their arguments are *independent* or *ground*

Rule 2) the arguments of a functional term are reduced in parallel when their WHNFs are demanded and the function is a strict one

Rule 3) the alternative clauses and rewrite rules are tried sequentially using the top-down strategy

---

The algorithm of *independent* and *ground* are same as the ones defined in [DeGroot 1984]. This principle can be expressed with an intermediate code, called CGE^+ (*Conditional Graph Expression*$^+$), which is an extension of DeGroot's CGE [DeGroot 1984]. It is used to express the necessary conditions to spawn the subgoals or function reductions in parallel. The body of a clause and right-hand side of a rewrite rule are expressed by the CGE^+, which is informally defined as follows;

---

1) G : a simple goal (or subgoal) whose argument can be a functional term.
2) $(SEQ\ E_1 \cdots E_n)$: execute expressions E_1 through E_n sequentially
3) $(PAR\ E_1 \cdots E_n)$: execute expressions E_1 through E_n in parallel
4) $(GPAR\ (V_1 \cdots V_k)\ E_1 \cdots E_n)$: if all the variables V_1 through V_k are ground, then execute expressions E_1 through E_n in parallel ; otherwise, execute them sequentially
5) $(IPAR\ (V_1 \cdots V_k)\ E_1 \cdots E_n)$: if all the variables V_1 through V_k are mutually independent, then execute expression E_1 through E_n in parallel; otherwise, execute them sequentially
6) $(IF\ B\ E_1\ E_2)$: if the expression B is evaluated to true, execute expression E_1; otherwise, execute expression E_2
7) $F(SEQ\ F_1 \cdots F_n)$: if F is a construct symbol or non-strict function symbol, then construct WHNF $F(F_1 \cdots F_n)$ sequentially ; otherwise (*i.e.* F is a strict function symbol) evaluate expressions F_1 through F_n sequentially and eventually evaluate $F(F_1' \cdots F_n')$
8) $F(PAR\ F_1 \cdots F_n)$: if F is a construct symbol or non-strict function symbol, then construct WHNF $F(F_1 \cdots F_n)$ sequentially ; otherwise (*i.e.* F is a strict function symbol) evaluate expressions F_1 through F_n in parallel and eventually evaluate $F(F_1' \cdots F_n')$

The expressions 1) through 6) are the same as the DeGroot's CGE for the clause (actually they are improved CGE defined in [Hermenegildo 1986]), while expressions 7) and 8) are new expressions for rewrite

rules. Note that there are no conditions to check the groundness of function arguments in the expressions 7) and 8), since they are automatically checked by pattern matching semantics of the rewrite rules. That is, the arguments of a rewrite rule are always ground, hence, they can be always evaluated in parallel. In expression 8), the arguments are reduced in parallel only if the function is strict, which results in its WHNF. Otherwise, it is rewritten to the term in the right-hand side, which is returned as the result. In this case, as it is not WHNF, it induces another reduction process. The reason to adopt this reduction strategy rather than directly call the non-strict function, is in order to keep the storage optimization based on tail-recursion.

Example 2 is a CGE^+ for a Lazy Aflog program. It can be automatically generated from the Lazy Aflog program by the parallelizing compiler, or programmed directly by the programmer.

Example 2 A CGE^+ for the Lazy Aflog program

C_1 : test(X,Y) :- (**IPAR** (X,Y) p(X,Z) q(Y,W)), r(f(Z), g(W)).
C_2 : test(X,Y).
C_3 : p(a,1).
C_4 : p(b,2).
C_5 : q(c,3).
C_6 : q(d,4).
C_7 : r(2,5).
C_8 : r(4,72).
F_1 : f(X) ==> (X == 0) | 0,
+(**PAR** fib(X) fib(2*X)).
F_2 : g(Y) ==> *(**PAR** factorial(Y), fib(Y)).

In Example 2, as the subgoals $p(X,Z)$ and $q(Y,W)$ would generate the values of Z and W that are taken into the terms $f(Z)$ and $g(W)$, the goal $r(f(Z),g(W))$ should be executed after the evaluation of them. Figure 2 is the snapshots of the parallel execution of the CGE^+ in Example 2 when "Q_1 :- test(b,d)" is given. In Figure 2, the rectangle, circle and rounded-rectangle represent OR node, AND node and reduction node, respectively. The number attached to each node represents the order of execution, while the filled nodes represent the activated nodes at that time. Note that, since the *Unification Parallelism* is not exploited in PR^3, the functional terms $f(1)$ and $g(3)$ in the step (c) are reduced sequentially, although they can be evaluated in parallel if the *innermost-like* reduction strategy is used. The backward execution of the PR^3 is the same as the one presented in [Hermenegildo 1986] because there are no backtracking in the reduction phases after a functional term is eventually reduced to WHNF. For example, in the step (d), the subgoal $q(Y,W)$ which generate the arguments W would search alternative solutions for Y and W when a fail is occurred, rather than to generate another WHNF for $g(3)$ or $f(1)$.

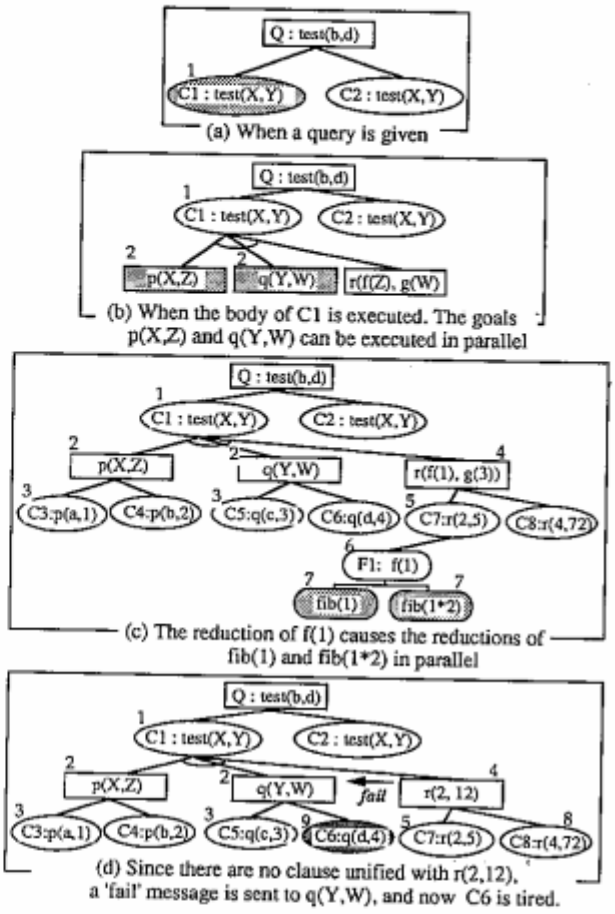


Figure 2 The Parallel Execution Snapshots of the Lazy Aflog Program in Example 2

4 A Parallel Extension of FWAM-II for PR³

The desirable characteristics of parallel abstract machine is to support the parallel execution while retaining the performance optimizations offered by the current sequential systems. To achieve this goal, a parallel abstract machine for PR^3, called PFWAM-II (Parallel FWAM-II), is designed as an extension of the sequential abstract machine FWAM-II. It is equipped with the run-time structures and instruction set to fork and join the parallel executions. We adopted the run-time structures and instructions of the RAP-WAM for the extension of FWAM-II because it is also an extension of WAM for AND-Parallel execution of Prolog and has a general primitive to fork and join the parallel tasks. Figure 3 shows the relationships between WAM, FWAM-II, RAP-WAM, and PFWAM-II.

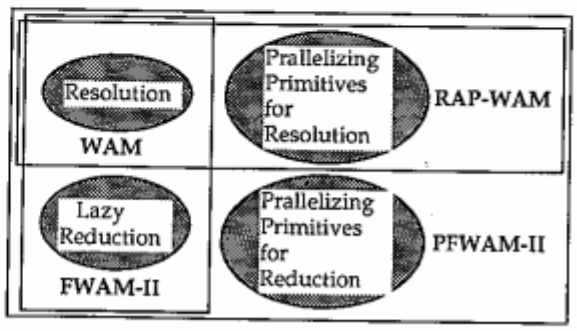


Figure 3 The Relationships Between WAM, FWAM-II, RAP-WAM and PFWAM-II

4.1 Run-Time Structures for Parallel Execution

The run-time structure of PFWAM-II is an extension of FWAM-II for parallel executions as shown in Figure 4. It consists of three parts ; First, the Heap, Trail, Environment, and Choice Point are structures for the execution of the logic part, and inherited from WAM ; Secondly, RS (Reduction Stack) is the structure only for the function reduction, and inherited from FWAM-II ; Finally, GS (Goal Stack), ParCall Frame, Local Goal Marker, Input Goal Marker, and Wait Marker are run-time structures for the parallel executions of subgoal or function reduction, that are inherited from RAP-WAM with slight modifications.

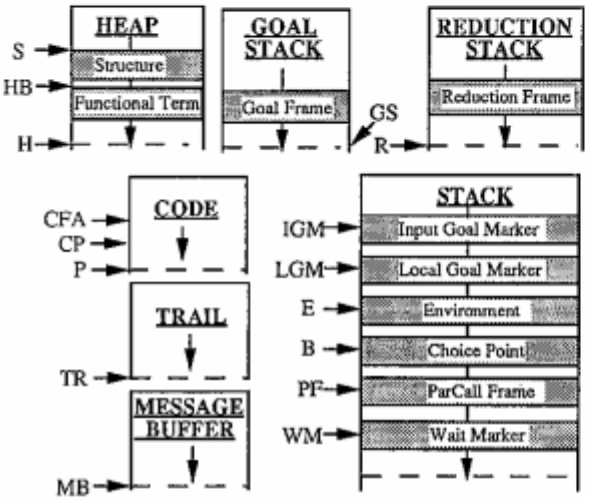


Figure 4 Data Areas and Registers for One PFWAM-II

In fact the run-time structures of the parallel execution is almost the same as that of RAP-WAM except that a parallel task in PFWAM-II can be a reduction of a functional term as well as the evaluation of subgoal, whereas in RAP-WAM, only the evaluation of a subgoal can be a parallel task. The run-time structure for parallel execution are the *Goal Frame, ParCall Frame, Input Goal Marker, Local Goal Marker,* and *Wait Marker.* Let us explain them focusing on the extensions which allow them to be also used for function reduction.

- *The Goal Frame* :
  The subgoals or the functional terms which are ready to be executed in parallel are pushed onto the Goal Stack. Each entry in the GS is also called a *Goal Frame* as in RAP-WAM. A *Goal Frame* contains all the necessary information for the remote execution of tasks. There are two kinds of *Goal Frame* in PFWAM-II; one is for a subgoal, and the other is for a function reduction. They are distinguished by the special tag in the *Goal Frame.* When a *Goal Frame* is the one for the subgoal, the structure of *Goal Frame* is the same as in RAP-WAM ; otherwise (*i.e.* it is one for the function reduction), it contains the extra pointer to the functional term to be reduced. In both cases, they are stolen from *Goal Stack* by a remote processor, and executed remotely in the same way.
- *The ParCall Frame* :
  It is used to keep track of the parallel tasks during forward and backward executions of PR^3. The entries and meanings of the ParCall Frame that is created for each parallel task are the same as in RAP-WAM. If a *ParCall Frame* is the one for the parallel function reductions, it immediately disappears from the *Local Stack* when the parallel reductions are completed because there is no backtracking in the reduction process. It is different from the case of parallel subgoal calls, in which it remains in the *Local Stack* in order to select the appropriate actions during backtracking.
- The entries and meanings of the *Input Goal Marker, Local Goal Marker,* and *Wait Marker* are the same as in RAP-WAM. However, they also immediately disappears when the task is a function reduction and it is reduced to WHNF.

The general execution scenario of PFWAM-II is as follows. As soon as a processor steals a task from another processor's *Goal Stack,* it creates an *Input Goal Marker* on its top of *Local Stack,* and checks whether it is a subgoal or a function reduction. If it is a subgoal, the processor starts working on the stolen subgoal by loading its argument registers from the parameter register fields in the *Goal Frame* and fetching instructions starting at the location (procedure address) received. If the stolen task is a function reduction, the processor loads the arguments and finds the starting address of the corresponding rewrite rule by referencing the functional term stored in the *Heap* of the parent processor. It was recorded on the *Goal Frame* by the parent processor. At any case, the local stacks of the processor will then grow (and shrink) as indicated by the semantics of FWAM-II.

When a parallel call is reached, a ParCall frame is created on the top of the *Local Stack* and tasks are pushed on to the *Goal Stack.* If there are no idle processors in the system at that time, the processor itself gets the goal from its *Goal Stack* again, makes a *Local Goal Marker,* and executes the task locally. If the parallel call is one for the subgoals, an *Wait Marker* is created on the top of the *Local Stack* as soon as all subgoals succeed. It is used for the backward execution of PFWAM-II. However, if the parallel call is for the function reduction, the *ParCall Frame, Local Goal Marker,* or *Input Goal Marker,* created on the local Stack can be removed since there is no backtracking in the reduction process. After the parallel call is finished, the execution can continue normally beyond the parallel call.

4.2 Instruction Set

The instruction set of PFWAM-II consists of the FWAM-II instructions and the new instructions implementing RAP as shown in Table-1. Since the FWAM-II instructions were explained in [Nang *et al.* 1991] and the instructions to fork and join the parallel call when tasks are subgoals are almost the same as the RAP-WAM, we only explain the instructions to control the parallel reduction. To fork and join the parallel executions are actually the same as the RAP-WAM when the parallel call is a determinate one. However, some attentions are required since the tasks to be forked can be functional terms.

- *push_reduce Vn, Slot_Num*
  It makes a new goal frame on the *Goal Stack* with the *Slot_Num* for the functional term pointed by Vn.

<Table-1> The PFWAM-II Instruction Set

The PFWAM-II Instruction Set					
WAM Instructions					
Procedure Control		**Indexing**		**Clause Control**	
try	L	switch_on_term	Ai, v,c,l,s	call	P/arity
retry	L	switch_on_constant	n, ff	execute	
trust	L	switch_on_structure	n, ff	proceed	
try_me_else	L			allocate	
retry_me_else	L			deallocate	
trust_me_else	fail				
Get		**Put**		**Unify**	
get_variable	Vi, Ai	put_variable	Vi, Ai	unify_variable	Vi
get_value	Vi, Ai	put_value	Vi, Ai	unify_value	Vi
		put_unsafe_value	Yi, Ai	unify_unsafe_value	Yi
get_constant	C, Ai	put_constant	C, Ai	unify_constant	C
get_list	Ai	put_list	Ai	unify_list	
get_structure	S, Ai	put_structure	S, Ai	unify_structure	S
get_nil	Ai	put_nil	Ai	unify_nil	
				unify_void	
Reduction Instructions					
Fget		**Matching**		**Writing**	
fget_value	Vi, Ai	match_value	Vi	write_value	Ai
fget_constant	C, Ai	match_constant	C	write_constant	C
fget_list	Ai	match_structure	S	write_structure	S
fget_structure	S, Ai	match_list		write_list	Ai
fget_nil	Ai			write_function	F, Ai
				write_structure_value	S, Ai
Reduction Control		**Rewriting**		**Reducing**	
commit		rewrite_value	Vi	reduce_value	Ai
return					
Parallel Abstract Machine Specific Instructions					
RAP-WAM Instructions				**Parallel Reduction Specifics**	
check_me_else_label	Label	push_call	Pid,Arity,Slot#	push_reduce	Vn, Slot_#
check_ground	Vn	check_ready	Slot_#, Label	deallocate_pcall	
allocate_pcall	#_of_slot, M	check_independent	Vn, Vm		
pop_pending_goal		waiting_on_siblings			

- *deallocate_pcall*
  It is used to join the parallel reductions. It waits until the number of goals to wait on in current *ParCall Frame* is 0; then, removes the current *ParCall Frame* from the local Stack.

Figure 5 shows the simplified PFWAM-II codes for F_2 of the CGE^+ in Example 2, in which since '+' and '*' are strict functions, their arguments are reduced directly rather than constructing the functional closure.

F_2 : g(Y) ==> *(**PAR** factorial(Y), fib(Y)).

```
F2:   allocate
      % Pattern Matching
      fget_value                    Y1, X1
      % Spawn Parallel Reduction for factorial(Y)
      allocate_pcall                2, 2
      put_value                     Y1, X1
      write_function                factorial/1, Y2
      write_value                   X1
      push_reduce                   Y2, 2
      % Spawn Parallel Reduction for fib(Y)
      put_value                     Y1, X1
      write_function                fib/1, Y3
      write_value                   X1
      push_reduce                   Y3, 1
      % Gather the Results
      pop_pending_goal
      deallocate_pcall
      % Construct WHNF
      put_value                     Y2, X1
      put_value                     Y3, X2
      call_P_Arity_N                */2, 2, 1
      rewrite_value                 X1
      % Returning
      return
```

Figure 5 An Compilation Example for CGE^+ in Example 2

5 Analysis

5.1 Performance Evaluation

In order to estimate the performance of our parallel extension, a simulator for PFWAM-II is developed. In this simulation, we assumed that there is a common shared memory for the run-time structures of each processor which are interconnected by a network. Each processor can access the run-time structures of other processors without additional overheads. The performance of PFWAM-II is estimated by counting the number of memory and register references, where the time for referencing data stored in the shared memory (whether it is local or not) is assumed 3 times longer than the time for register referencing, and the times for other operations such as arithmetic are ignored for the sake of simplicity.

We use three benchmark programs : the first one is *Fibonacci10* that is to compute the 10th fibonacci number, the second is *Check50* [Hermenegildo 1986] in which there are 10 parallel tasks each of which calls itself 50 times, and the third is *Symbolic Derivation* [Hermenegildo 1986] which is to find the derivative with respect to a variable. There are 176 parallel tasks in the *Fibonacci10*,

10 parallel tasks in the *Check50*, and 152 parallel tasks in the *Symbolic Derivation*. These benchmarks are programmed in both of logic and functional programming. In the simulation of function reduction, the effect of different reduction strategies is also measured. The simulated reduction strategies are *Innermost Reduction* in which the innermost functional terms are reduced first before the outer is tried, *Semi-Lazy* in which only the strict functions are reduced in the innermost fashion, and *Lazy Reduction* in which all functions are reduced in the outermost fashion.

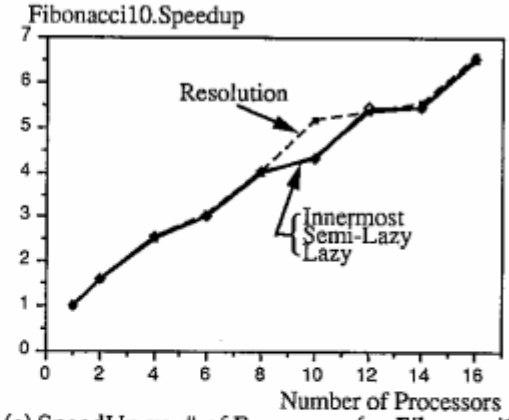
Upon the simulation results, the parallelizing overhead, which is defined as the extra execution time for parallel code running on the single processor, is measured as about 30~60 % when the grain size is relatively small (for example, *Fibonacci10* and *Symbolic Derivation*), whereas about less than 1 % when the grain size of parallel task is large enough to ignore the overhead (for example, *Check50*). Figure 6 graphically shows the speedup of the execution time of all benchmark programs as a function of the number of processors. In this figure, since *Check50* has only 10 parallel tasks, the speedup doest not increase when the number of processors is larger than 10. The speedup of other benchmark programs are not linear because they have too fine-grained parallelism. The most important fact which can be identified from Figure 5 is that, *whether they are programmed in the logic or functional style, and whether the reduction strategy is innermost or outermost, the speedup behaviour is almost same.* The speedup ratio is not dependent on the execution mechanisms, but the availability and grain size of parallelism in the benchmark programs. In other words, PFWAM-II can support both of the parallel resolution and parallel reduction with the almost same efficiency.

Figure 7 shows the Working, Waiting, and Idle times for *Symbolic Derivation* as a function of the number of processors. It is from identified from Figure 7 that the processor utilization ratio is reduced proportional to the number of processors, and the parallel reduction mechanism permits higher utilization ratio than parallel resolution because there is no restriction to steal a task from other processors when the task is a function reduction (*i.e.*, there is no "garbage slot problem [Hermenegildo 1986]" when executing the function reduction).

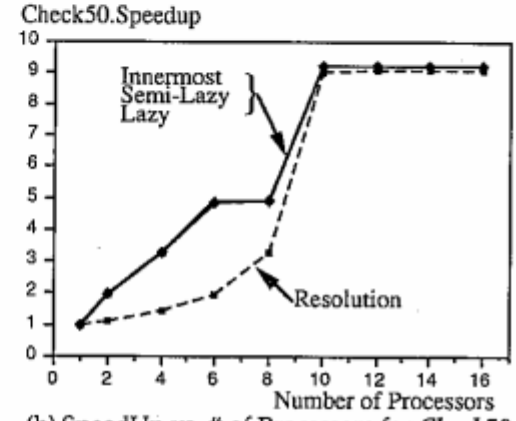
5.2 Comparison with Related Work

One of the most related works is the CSELT's work centering around K-LEAF. K-LEAF [Levi and Bosco 1987] is a functional logic language based on the transformation. A rewrite rule in K-LEAF program is transformed into Prolog clause with an extra argument for the return value, and the nested function is flattened with *produced variable* for the outermost search strategy. K-WAM is an abstract machine to support outermost-SLD resolution which is the inference rule of K-LEAF. Accordingly, there is no real reduction mechanism in K-LEAF and K-WAM.

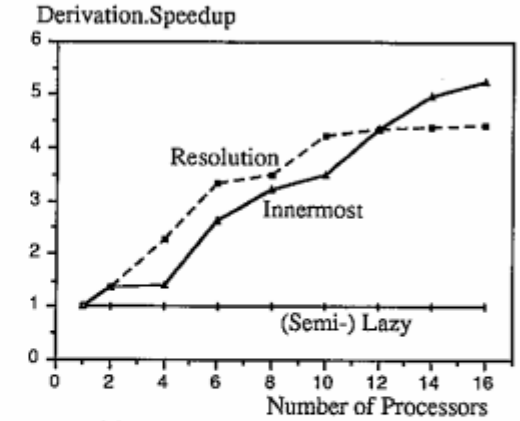
A parallel extension of K-WAM on a distributed memory multiprocessor is also developed [Bosco *et al.* 1990]. In this work, K-WAM is extended to control the OR-parallel execution of K-LEAF programs, and AND-parallelism is restricted to be *one-solution*. The major difference between the parallel extension of K-WAM and PFWAM-II is that the former is designed for exploiting only OR-parallelism in the flattened K-LEAF programs, while the later is designed for exploiting only AND-parallelism of Lazy Aflog programs.



(a) SpeedUp vs. # of Processors for ***Fibonacci10***



(b) SpeedUp vs. # of Processors for ***Check50***

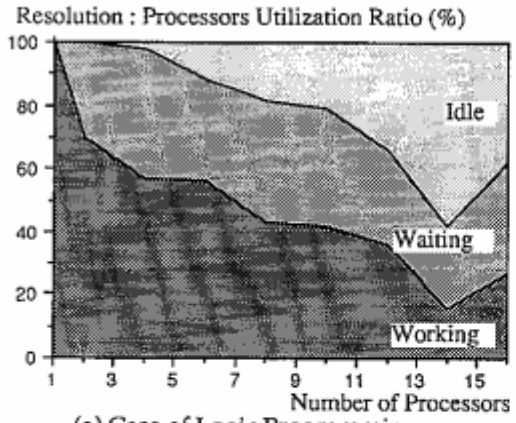


(c) SpeedUp vs. # of Processors for ***Symbolic Derivation***

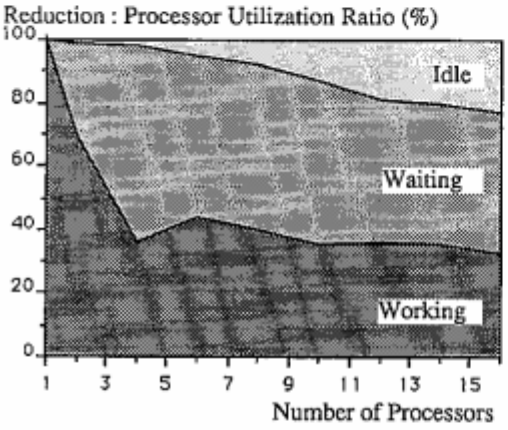
Figure 6 SpeedUp vs. Number of Processors for Benchmark Programs

6 Summary

This paper presents a pair of a parallel computational model and its abstract machine for a functional logic language, called Lazy Aflog, which was proposed as a cost-effective mechanism to incorporate functional language features into logic language. The proposed



(a) Case of Logic Programming
(Parallel Resolution)



(b) Case of Functional Programming
(Parallel Reduction)

Figure 7 Working, Waiting Idle Times for *Symbolic Derivation*

computational model underlies DeGroot's RAP model because the Restricted-AND Parallelism could be easily exploited in both of the function and logic. However, some extensions are required since there is a parallel function reduction in the functional part of Lazy Aflog programs. Since RAP-WAM includes the general structures to fork and join the parallel tasks, the parallel function reductions can be also supported efficiently with slight extension. A parallel abstract machine based on the RAP-WAM and extension of FWAM-II, called PFWAM-II, is also proposed as an implementation method on a multiprocessor. Several simulation results show that PFWAM-II can support not only the parallel resolution, but also parallel reduction with the almost same efficiency.

References

[Bellia and Levi 1986] M. Bellia and G. Levi, The Relation between Logic and Functional Languages : A Survey, *Journal of Logic Programming*, Vol. 3, No. 3, 1986, pp. 217-236.

[Bosco *et al.* 1989] P. G. Bosco, C. Cecchi, C. Moiso, An Extension of WAM for K-LEAF : a WAM-based compilation of conditional narrowing, *Proc. of 6th Int'l Conference on Logic Programming*, MIT Press, 1989, pp. 318-333.

[Bosco *et al.* 1990] P. G. Bosco, C. Cecchi, C. Moiso, M. Porta, and G. Sofi, Logic and Functional Programming on Distributed Memory Architectures, *Proc. of 7th Int'l Conference on Logic Programming*, 1990, pp. 325-339.

[Conery 1983] J. S. Conery, *The AND/OR Process Model for Parallel Execution of Logic Programs*, PhD These, Univ. of California, Irvine, 1983.

[DeGroot 1984] D. DeGroot, Restricted AND-Parallelism, *Proc. of FGCS84, ICOT, 1984, pp. 471-478.*

[Hermenegildo 1986] M. V. Hermenegildo, *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*, TR-86-20, Dept. of Computer Science, Univ. of Texas at Austin, 1986.

[Levi and Bosco 1987] G. Levi and P. G. Bosco, A Complete Semantic Characterization of K-LEAF, A Logic Language with Partial Functions, *Proc. 1987 Symposium on Logic Programming*, IEEE Computer Society Press, 1987, pp. 318-327.

[Nadathur and Jayaraman 1989] G. Nadathur, B. Jayaraman, Towards a WAM Model for λProlog, *Logic Programming : Proceedings of the North American Conference*, MIT Press, 1989, pp. 1180-1198.

[Nang *et al.* 1991] Jong H. Nang, D. W. Shin, S. R. Maeng, and J. W. Cho, An Effective Incorporation of Function into Logic, *Information and Software Technology*, Vol 33, No. 8, Butterworth Heimann Ltd., U.K, 1991.

[Nang 1992] Jong H. Nang, *A Study on the Execution Mechanisms of a Functional Logic Language with Lazy Evaluation*, PhD Dissertation, KAIST, 1992.

[Peyton Jones 1986] S. L. Peyton Jones, *The implementation of functional programming languages*, Prentice-Hall, 1986.

[Peyton Jones *et al.* 1987] S. L. Peyton Jones, C. Clack, J. Salkid, and M. Haridi, GRIP - A High Performance Architecture for Parallel Graph Reduction, *Proc. of Functional Programming Language and Computer Architecture*, Springer-Verlag LNCS 274, 1987, pp.98-112.

[Shin *et. al* 1987] D. W. Shin, J. H. Nang, S. Han, and S. R. Maeng, A Functional Logic Language Based on Canonical Unification, *Proc. 1987 Symposium on Logic Programming*, IEEE Computer Society Press, 1987, pp. 328-334.

[Shin *et al.* 1988] D. W. Shin, J. H. Nang, S. R. Maeng, and J. W. Cho, The Semantics of a Functional Logic Language with Input Mode, *Proc. of the International Conference On the Fifth Generation Computer Systems 1988*, 1988.

[Shin *et al.* 1992] D. W. Shin, J. H. Nang, S. R. Maeng, and J. W. Cho, A Functional Extension of Logic Programming, *New Generation Computing*, 1992 (accepted for publication).