A VHDL Specification of a Shared Memory Parallel Machine for Babel*

W. Hans†, J.J. Ruz‡, F. Sáenz‡, S. Winkler†

† RWTH Aachen, Lehrstuhl für Informatik II, D-52056 Aachen, Germany, {hans,winkler}@zeus.informatik.rwth-aachen.de
‡ Universidad Complutense de Madrid (UCM), Departamento de Informática y Automática, 28040 Madrid, Spain, {fernan,jjruz}@dia.ucm.es, Fax # : +34-1-3944687, Tel # : +34-1-3944356

Abstract

We present an abstract machine designed for the parallel execution of functional logic programs, i. e. Babel. It is accomplished utilizing a shared memory model. Efficiency is gained by using the same stack mechanisms as the WAM, i.e. the fast reclamation of memory during backtracking is maintained despite the parallel extensions.

In addition to the strict behaviour of programs (e.g. in Prolog), Babel offers non-strict functions that do not necessarily demand all of the arguments for the evaluation. We study the work load behaviour and the introduced restrictions imposed by the insistence on the fast stack mechanisms. The specification is carried out in the hardware description language VHDL offering simulation facilities, and having a temporal model which allows to gain a performance study of the parallel system. We present the results obtained from the simulation of the VHDL specification, showing the speed-ups gained with different numbers of processors.

1 Introduction

Functional programming languages are based on equations and the lambda calculus. They offer higher order functions and a strong type system. Their execution model is based on the *reduction principle*. The operational semantics can be eager (*innermost* reduction) or lazy (*outer* reduction). Innermost reduction tries to call each function when all of their arguments are evaluated. Outer reduction only demands argument evaluation when they are actually needed.

Logic programming languages rely on the first order predicate logic; they offer the power of logical variables, unification, and nondeterminism. Their (more complex) execution model is based on *resolution*. Operational semantics of Prolog (the best known representative of logic programming languages) is based on SLD-resolution.

Many languages have been proposed for combining the advantages of functional and logic programming languages [19, 4, 1, 16, 2, 9]. While logic functional programming extends logic programming with functional features [4], functional logic programming extends functional programming with first order logical variables and nondeterminism [19].

*This work was supported by the Spanish PRONTIC project TIC92-0793-C02-01 and by the German DFG-grant In 20/6-1.

Babel [19] is a functional logic programming language that embodies both the functional and logical paradigms. Its operational semantics is based on *narrowing*, a mechanism that subsumes SLD-resolution and reduction by performing rewriting on terms, and unification.

The extension to a parallel execution model for an innermost reduction mechanism is feasible because argument evaluation can be achieved in parallel due to the independence between arguments and the strictness of the functions in all the arguments [1].

One representative parallel model for logic programming is the independent And parallel execution model (IAP) [11], which is based on the restricted And parallel execution model (RAP) [8]. IAP exploits the parallel execution of goals provided that they are independent. Two goals are independent if they do not share logical variables. This restriction is needed to avoid the problems that arise when a logical variable is instantiated to different values.

Several parallel execution models have been proposed for the combination of both functional and logic programming paradigms [13, 22, 15]. All of them adapt the graph-based sequential machine [14] to a parallel system. Our approach incorporates a more sophisticated memory management [17] relying on the stack-based mechanism widely used in other implementations (e.g. in the WAM [23]). Furthermore, we follow the strict parallel model [21] applying the concepts of And-parallelism embodied in logic programming to functional logic programming. Moreover, we modify the non-strict parallel model of [21] in order to develop the suitable semantics.

Different concepts have been proposed for implementing functional logic languages. Beginning with graph-oriented narrowing in the functional way [14], other more efficient approach has been proposed [17]. The advantages of the latter approach rely on the fact that most of the efficient techniques present in the Warren abstract machine (WAM) [23] are retained in the design of the functional logic system.

In this report a parallel stack-based non-strict model for Babel is developed and refined through several stages. We firstly sketch the parallel model informally and point out the ideas of the stack-based model. Finally we validate both the specification and the machine by using the hardware description language VHDL. This language provides a well suited specification tool that allows the functional validation of the parallel system, too. Moreover, since VHDL has a temporal model, is is possible to simulate at a very low-level stage the behaviour of the system. We have designed a basic shared memory system which allows us to take measurements concerning memory access timings. We compute the speed-ups of a benchmark program by means of the simulation of the specification, which yields the absolute timings regarding the execution of the program running on a given number of processors. Since we obtain timings close to an actual parallel system, we are able to take relevant design decisions for optimization purposes and even for the development of special hardware able to run parallel Babel programs.

This paper is organized as follows: Section 2 briefly presents an innermost first order version of Babel. In Section 3, the exploited source of parallelism is introduced, giving a preliminary view of the computational mechanism. Section 4 presents the stack-based parallelism model. The architecture of a parallel machine together with the instruction set and the compilation scheme is presented in Section 5 which coincides with the proposed parallelism model. The hardware description language VHDL is used in Section 6 to specify and validate the parallel machine. Section 7 sketches the first preliminary results

[1] A function f is said to be strict in its i-th argument iff $f(a_1, \ldots, a_i, \ldots, a_n)$ is undefined whenever a_i is undefined.

of the parallel system running some selected benchmarks. Finally, Section 8 summarizes the conclusions and points out the direction future work may take.

2 The Babel Language

Babel is a functional logic language with a constructor discipline [2] and a polymorphic type system. It has a functional syntax and uses narrowing [20] as its evaluation mechanism. For the sake of clarity, we will consider the first order subset of Babel with the leftmost innermost narrowing strategy applied [3].

A Babel program consists of a finite sequence of function definitions and can be queried with a goal expression. Each function f is defined by a finite sequence of rules, where each rule has the form:

$$\underbrace{f(t_1,\ldots,t_n)}_{\text{left hand side } (lhs)} \quad := \quad \underbrace{\underbrace{\{B\rightarrow\}}_{\text{optional guard}} \quad \underbrace{M}_{\text{body}}}_{\text{right hand side } (rhs)} \ .$$

where B is a Boolean expression, and M is an arbitrary expression. The rules must satisfy the following conditions:

1. *Left linear data pattern:* t_i are data terms, that share no variables, altogether.
2. *Restrictions on free variables:* any variable that occurs only in the *rhs* is called *free*. Free variables are allowed to occur in the guard, but not in the body.
3. *Non ambiguity:* Babel functions are (partial) functions in the mathematical sense, i.e. for each tuple of (ground) arguments, there is at most one result. This is ensured by special syntactic restrictions (see [19] for details) [4].

A *term* t is either a (logical) *variable* (denoted by an identifier beginning with a capital letter) or an application of a n-ary data constructor $c \in DC^n$ to n argument terms:

$t ::= X$	% variable
$\mid c(t_1 \ldots t_n)$	% application of a n-ary data constructor c

An expression $M \in Exp$ has the form:

$M ::= X$	% variable
$\mid \varphi(M_1,\ldots,M_n)$	% φ is a n-ary function or constructor symbol
$\mid B \rightarrow M_1\{\Box M_2\}$	% if B then M_1 else undefined { else M_2}

Several built-ins such as conjunction ($\wedge$), disjunction ($\vee$), and Boolean negation ($\neg$) are also supported. Negation is defined by:

$$\neg false := true. \quad \neg true := false.$$

The language also includes weak equality defined by the following rules:

[2] Constructor discipline means that constructor and function symbol are distinguished although it is possible to match constructor symbols with uninterpreted function symbols.

[3] The extension to the higher order version introduces no restrictions.

[4] We are allowed to remove the condition about *left linearity* in the eager semantics because there is no delay of the unification to the *lhs*, as happens in an unification-by-demand scheme.

$$
\begin{array}{lll}
(c = c) & := true. & \%\ c \in DC^0 \\
(c(X_1,\ldots,X_n) = c(Y_1,\ldots,Y_n)) & := (X_1 = Y_1) \wedge \ldots \wedge (X_n = Y_n). & \%\ c \in DC^n, n > 0 \\
(c(X_1,\ldots,X_n) = d(Y_1,\ldots,Y_m)) & := false. & \%\ c \in DC^n, d \in DC^m \\
 & & \%\ c \neq d, \vee n \neq m
\end{array}
$$

We consider the operational semantics defined by the eager narrowing mechanism that evaluates function and constructor arguments in a leftmost innermost order, except the built-in *non-strict* functions (conditional, biconditional, conjunction, and disjunction).

Let $M \equiv f(M_1,\ldots,M_n)$ be an expression and let $f(t_1,\ldots,t_n) := M'$ be a variant of a rule sharing no variables with M. Moreover, let $\sigma \circ \lambda$ be the *most general unifier* of M and $f(t_1,\ldots,t_n)$, i.e. the minimal substitution (of variables by expressions) such that $M\sigma$ is syntactically identical to $f(t_1,\ldots,t_n)\lambda$. Then, M can be *narrowed* (in one step) to $M'' \equiv M'\lambda$ with *answer substitution* σ (denoted by $M \Longrightarrow_\sigma M''$).

In order to cope with the built-in non-strict functions, the basic narrowing mechanism is extended with the following narrowing rules:

$$
\begin{array}{lll}
(false \wedge B) \Longrightarrow_\varepsilon false & (true \vee B) \Longrightarrow_\varepsilon true & \\
(true \wedge B) \Longrightarrow_\varepsilon B & (false \vee B) \Longrightarrow_\varepsilon B & \\
(true \rightarrow M) \Longrightarrow_\varepsilon M & (true \rightarrow M_1 \Box M_2) \Longrightarrow_\varepsilon M_1 & (false \rightarrow M_1 \Box M_2) \Longrightarrow_\varepsilon M_2
\end{array}
$$

Where ε denotes the empty substitution ($\varepsilon(X) = X, \forall X \in Var$).

The one step narrowing relation $\Longrightarrow_\sigma \subseteq Exp \times Exp$ where $\sigma : Var \rightarrow Exp$ is canonically extended to arbitrary expressions:

- $\dfrac{M_i \Longrightarrow_\sigma N_i}{\varphi(M_1,\ldots,M_n) \Longrightarrow_\sigma \varphi(M_1\sigma,\ldots,N_i,\ldots,M_n\sigma)}$
- $\dfrac{B \Longrightarrow_\sigma B'}{(B \rightarrow M_1\{\Box M_2\}) \Longrightarrow_\sigma (B' \rightarrow M_1\sigma\{\Box M_2\sigma\})}$

The narrowing sequence of an expression M is denoted by the transitive reflexive closure $\overset{*}{\Longrightarrow}_\sigma$ of $\Longrightarrow_{\sigma'}$ with the composition of substitutions. A sequence of steps $M \Longrightarrow_{\sigma_1} \ldots \Longrightarrow_{\sigma_m} M'$ is called a *computation* of the goal M with substitution $\sigma := \sigma_1 \circ \ldots \circ \sigma_m$. The result of the narrowing can yield:

- *success:* $M \overset{*}{\Longrightarrow}_\sigma t$, $t \in Term$ with *answer substitution* σ,
- *failure:* $M \overset{*}{\Longrightarrow}_\sigma N$, $N \notin Term$ and $\Longrightarrow_\sigma$ cannot be applied to N anymore, or
- *non-termination.*

In the following we consider only the (sequential) leftmost innermost narrowing strategy. This means that expressions at leftmost innermost positions are narrowed first. When dealing with conjunctions and disjunctions we call *false* and *true* definitory values, respectively, because such a result of one operand determines the outcome of the whole con-/disjunction.

Example 1. A Babel function definition for *append*. The classical procedure for *append* is defined in Babel:

$append([], Xs) \ := \ Xs.$
$append([X|Xs], Ys) \ := \ [X|append(Xs, Ys)].$

It can be used in a Prolog-like fashion, because of the use of logical variables. Therefore, we can use it either to append two lists, to compute list differences, or to test sublists of lists. Moreover, Prolog programs can be straightforwardly translated into Babel programs. For instance, the following Prolog procedure:

$append([], Xs, Xs).$
$append([X|Xs], Ys, [X|Zs]) \; :- \; append(Xs, Ys, Zs).$
is translated into:
$append([], Xs, Xs) \; := \; true.$
$append([X|Xs], Ys, [Z|Zs]) \; := \; Z = X \wedge append(Xs, Ys, Zs) \rightarrow true.$

Prolog notation is allowed in Babel as a syntactic 'sugar' of the functional definition.

3 Exploiting And-parallelism

Several sources of parallelism for Babel can be considered: independent And parallelism [8, 11], Or parallelism [24], Stream parallelism [6], and Unification parallelism [7] for logic programming. Independent And parallelism consists of the parallel execution of independent goals in a clause. Or parallelism consists of the parallel execution of clauses in a procedure. Stream parallelism exploits the parallelism of goals that share common variables in a producer-consumer scheme. Unification parallelism performs parallel unification of the arguments of the literal being solved with the corresponding clause heads. We will focus our attention on a suitable form of independent And parallelism for the functional logic programming paradigm. Our aim is the parallel execution of functional arguments on the right hand side of the rules. The term And parallelism originates from Prolog and denotes the kind of parallelism which is exploited for the arguments (literals) of the conjunction that represents the clause body. We adopt this kind of parallelism for Babel and apply it not only to the conjunction but also to all the other built-ins and user defined functions. Although another notion, e.g. subexpression parallelism, could be more adequate, we still call it And-parallelism in order to follow the previous nomenclature [13, 22, 15].

To start with, we present the topics related to the parallel computational mechanism surrounding the strict functions. Later, we extend this mechanism in order to cope with the built-in non-strict functions.

3.1 Parallel Computational Mechanism for Strict Functions

As mentioned above, we are interested in the parallel execution of arguments belonging to right hand sides of rules. An expression M defining the *rhs* of a rule has, in general, functional arguments (subexpressions with a function symbol as the root node). The parallel evaluation consists of the parallel execution of those functional arguments which meet the following independence conditions:

- the functional arguments are *independent*, i.e., two arguments are independent if they do not share any variables at run time.
- one of the functional arguments is not a descendant in the syntactic tree of another functional argument.

The first condition avoids the problems that arise when a variable, common to several expressions, is bound to different values simultaneously. Therefore, if a variable is bound to a ground term at a given program point, the expressions to which that variable belongs may become independent. This is the typical condition imposed in the And parallelism model (see [8, 11]).

The second condition states that a functional argument cannot be executed in parallel with one of its proper subexpressions. This is imposed in order to avoid the *suspension* that occurs when the functional argument requires the outcome of its syntactical son. Such *suspensions* imply a run-time management of synchronization between nodes in the syntax tree, therefore resulting in an extra overhead in the exploitation of parallelism. With our proposal, this kind of synchronization is restricted to be identified and annotated at compile-time.

In [13], a machine based on the unrestricted And parallelism is presented which is capable of evaluating functional arguments and their own subexpressions in parallel. Therefore, there is no need for both the restrictions. [15] presents a system in which the second condition is partially removed, a functional argument and its children are allowed to be executed in parallel. The experimental results of these systems will show whether the extra overhead is worthwhile.

In the following example we can see how these conditions are applied to a particular case.

Example 2.

Consider the recursive rule for flattening lists:

$$flatten([H|T]) \ := \ append(flatten(H), flatten(T)).$$

We say that *append* (strictly) depends on $flatten(H)$ and $flatten(T)$ because they are functional arguments of *append*. We say that $flatten(T)$ (conditionally) depends on $flatten(H)$ because they can be executed in parallel if H and T can be proved to share no variables.

The system presented in [21] generates the following parallel rule.

$$\begin{aligned} flatten([H|T]) := {} & let\ cpar(indep(H,T), \\ & \qquad A_1 := flatten(H),\ A_2 := flatten(T)) \\ & in\ append(A_1, A_2). \end{aligned}$$

It means that $flatten(H)$ and $flatten(T)$ will be executed in parallel if H and T are independent (first condition), otherwise a sequential execution will be performed. Anyway, *append* must wait for their computed results (second condition).

Next, we will describe briefly both the forward and the backward computational mechanism.

3.1.1 Forward Computation

Let $\varphi(M_1, \ldots, M_n)$ be an expression belonging to the *rhs* of a rule, assuming the parallel execution of the arguments $M_i (1 \leq i \leq n)$. If successful results are computed for all the arguments, then we have successfully reached the *join point* of the parallel call.

3.1.2 Backward Computation

Let us suppose that a failure is computed for the i-th argument position in the first parallel call invocation (i.e., *inside state* [10]). Then we are allowed to discard the current parallel computation, since there is no option to deliver any successful computation.

If the backtracking course reaches a previously computed parallel call (i. e. *outside state*) then we look for the rightmost argument position i with pending alternatives. The arguments to the right of i are reset and a new solution is requested from i. In the meantime, arguments to the right of i are spawned in parallel, anticipating subsequent work.

3.2 Parallel Computational Mechanism for Non-strict Functions

In this section we extend the basic strict parallel model in order to cope with the class of functions that can yield a definitory value, even if some arguments are undefined. The logical conjunction and disjunction, together with the conditional and biconditional functions belong to that class. For instance, the declarative semantics of the conjunction when one of the arguments yields *false* is also *false*, whatever the results of the remaining arguments are. The way in which the operational semantics of such non-strict functions is defined determines the operational behaviour of the system. At this point, we may consider two possible approaches.

In the first approach we can think of a scheme such that the (independent) arguments of the non-strict functions are allowed to be executed in parallel until one of them returns a definitory result. When this is achieved, we discard the outcomes of the remaining arguments and continue with the next narrowing step. In this way we can obtain solutions that cannot be achieved in the left-to-right sequential order, since we can consider the case in which an argument to the left of a definitory position is undefined (i.e., a failure). This scheme implies that the evaluation order of arguments is not fixed, and moreover becomes unpredictable. Although we occasionally obtain more solutions, there is the risk of running into an unexpected infinite computation branch, and thus becoming unable to compute the remaining solutions.

A more conservative approach relies on both getting the same solutions as the sequential scheme and maintaining the sequential order of solutions. In this way nondeterministic computations are avoided, allowing sequential-like nondeterminism. To achieve this goal we must pay attention to the definitory argument positions[5], selecting the leftmost one and discarding the results of the arguments to its right. Only the bindings up to the selected argument are kept in the success substitution and the others are discarded.

We develop the parallel system considering the second approach. and discuss both the forward and the backward computation.

3.2.1 Forward Computation

Let $\varphi(M_1, \ldots, M_n)$ be a non-strict function. As soon as a definitory result for the i-th argument is computed we pay attention to the arguments to the left of M_i. It may be the case that the arguments have already been computed, or not all of them have been computed. If the latter case holds, further definitory values may be computed. Eventually when all of the arguments to the left of the leftmost definitory argument position j

[5]The argument positions delivering a definitory value.

have been computed, we are committed to the bindings of arguments $M_1, \ldots, M_j$, discarding those of $M_{j+1}, \ldots, M_n$. If there is no definitory value at all, we simply wait for all arguments as in the sequential case.

3.2.2 Backward Computation

Let us consider that a failure is computed for the i-th argument position in *inside state*. We must wait for the arguments to the left of i before we backtrack outside of the parallel call, if some of them return definitory values. Then if the arguments to the left of the definitory argument have returned non-failed, non-definitory values, we simply continue with forward execution (out of the parallel call). Otherwise, we backtrack out of the parallel call.

If the backtracking course reaches an *outside state* we look for the rightmost argument position i with pending alternatives. The arguments to the right of i are reset and a new solution for i is requested. In the meantime, the arguments to the right of i are spawned in parallel. From this point, the behaviour is as the forward computation. But in the case of failure we always look for the rightmost argument with alternatives to try a redo.

The previous scheme explores the sequential narrowing path ensuring that all the sequential solutions are found in the expected order.

4 An Efficient Stack-based Model to Support Parallelism

One of the most important features of the Warren abstract machine (WAM) [23] is the efficient management of data structures. For instance, deallocation of frames belonging to the run-time stack or to the heap is done simply by altering the contents of machine registers. This behaviour promotes the backtracking mechanism because space recovery is very fast. During the forward computation, the data necessary to support nondeterminism are stored in the data areas (choice point frames and trail). In this way, whenever a clause in a procedure yields failure, another can be tried by restoring the machine registers previously saved in the corresponding choice point and by resetting the unifications due to the failing clause. In restoring the machine registers, the amount of heap allocated by the failing clause is automatically recovered because the data allocated on the heap always accumulate on top of the heap before clause execution. Moreover, environments and choice points allocated on the run-time stack exhibit the same behaviour. Finally, the respective bound variables are reset. Their addresses are known because they are pushed onto the trail whenever an unification affects them. The trail is also deallocated in the same way as the frames in the run-time stack. The resetting of the trail frame allocated for the failing clause is as straightforward as above, it simply consists of restoring the content of the trail register to the value stored in the choice point. All of this is possible because the order of the frames allocated in the stacks is maintained. In fact, if we push new frames on top of the old ones, we can be sure that only the corresponding frames will be deallocated on backtracking.

Our aim is to preserve as much as possible the features of the WAM. We will show how the stack-based model of the WAM can be extended to the parallel model of the functional logic language Babel. To do this, we will rely on the previous work for the stack-based implementation of (sequential) narrowing [17]. In that work, the same premise is set: only new frames will be allocated on top of older ones. The time notion of newer and older frames corresponds to a time order of nodes in the proof tree in the case of Prolog and the order of nodes in the narrowing tree. The order is therefore based on the resolution order and on the narrowing strategy. Since we consider the eager narrowing strategy, which follows a leftmost innermost strategy, a node is older than another whenever it is to the left or above, in the search path of the narrowing tree. A recursive definition of the age of every node in the narrowing tree can easily be defined. The mapping of the narrowing tree to the structures of the stack-based implementation gives implicit annotations of the relative age, since each new environment or choice point is always allocated upon older ones. In this way, the backtracking mechanism will always find the newest choice point to try new alternatives on top of the run-time stack.

To perform the narrowing of a (parallel) expression, the computation of the independent subexpressions can be delegated to several narrowing machines. Each parallel narrowing machine has several data areas available to perform the narrowing: the run-time stack, the heap, the trail, and the data stack. Whenever a fork point is reached in the narrowing, other machines are allowed to narrow one of the available parallel siblings. The result of the computation will be combined, at the join point, with the outcomes of each machine. On the basis of these considerations, we discuss the extensions of the sequential stack-based scheme to the parallel one.

Our aim is to keep a safe order of frames (trail entries, heap entries, data stack entries, environments, choice points and other data structures for parallelism support) inside the data areas so that only newer frames are allocated on top of older ones. At first sight, we restrict ourselves to consider only choice points, environments, and the data structures necessary for parallelism support in order to keep the presentation clear.

Let us suppose that up to a given point, the narrowing of an expression is pure sequential (without any previous parallel call) and therefore, since we retain the sequential behaviour of the parallel system for sequential constructors, the order of frames in the run-time stack is from older to newer following its growth. Then, we reach a fork point in which several parallel siblings are available. Let us also assume that the remaining machines have empty run-time stacks, and therefore any subexpression of the parallel call can be safely allocated in their own data areas. Depending on the available resources, all of the parallel siblings or only a subset may be picked up. The parent machine (in which the fork point occurred) will spawn the parallel siblings, but in order to remain busy until the parallel siblings have been computed by the remote machines, the oldest subexpression is locally narrowed. Now, if there are more available siblings at the fork point once the oldest one has been locally computed, we are able to continue with the oldest available sibling, since in this way we will push only newer frames on top of the local run-time stack. Meanwhile, some remote machines may have finished their work and sent the result to the parent. These new idle machines will look for work in such a way that only a frame newer than the topmost one will be allocated on top of their own stacks. Note in this point that the time order notion is not the sequential one anymore. In order to arrange ideas about the time order notion we will present how the parallel narrowing computation can be represented by means of a suitable scheme.

Each intermediate state of a sequential narrowing computation can be represented by means of an And-Or tree, in which *And* and *Or* nodes alternate (see Figure 1). An And node represents the head of a rule whose children are the applications of the right hand side of the rule. These applications are the Or nodes whose children are the heads of the rules that are unifiable with each Or node. The construction of such a tree is guided by the eager computation mechanism. A partial ordering is defined regarding that a frame a is newer than a frame b whenever b is found first in the narrowing tree in a leftmost innermost order. The position of a node in the tree definitively denotes its age.

Furthermore, each intermediate state of a parallel narrowing computation can be represented by adding to the previous scheme *Fork* nodes, which represent the parallel scheduling of their child Or nodes, and *Join* nodes, which represent join points. In this scheme, Or nodes as well as Fork nodes may be children of a given And node. Of course, we are able at this point to definitely associate a relative age between Or nodes and its children, as well as between children of Fork nodes. We can also set relative ages between the children of a given And node, provided they may be Fork or Or nodes. But we are not able to compare any ancestor node with descendant nodes of a Fork node until the Join node is reached, because the corresponding subtree configuration for the Fork node is only known after the join point. Note that, in general, several subtrees may be tried by the backtracking mechanism before reaching the successful configuration.

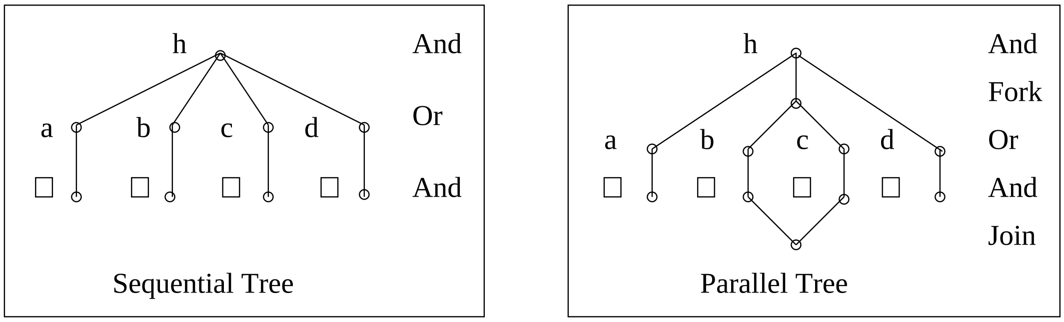

Figure 1: Sequential and Parallel Execution Trees

The above discussion hints how the proper parallel siblings are determined for the remote computation. Since we are interested in stealing newer parallel siblings so that the time order and therefore the safe deallocation of stack frames is preserved, we focus on the non-complete partial order hinted above. Therefore, when building the parallel narrowing tree, we know the definitely relative ages and the unknown ones, so that we can decide what parallel siblings are suitable to be processed by a given machine.

This precedence restriction implies that a machine in the parallel system may run out of work and can not steal work until the corresponding join point is reached. Several solutions may be taken in this point, the simplest one is to get idle. Another one is the creation of new machines. Further analysis of the parallel system will yield with suitable approaches.

Next, we consider in more detail the parallel stack-based model in order to prepare the detailed formal specification. In this stage we focus on the presentation of the aspects concerning the management of parallelism and delay the sequential narrowing computation to the next section. In the following we present some topics covered by the formal specification which is given at the end of this section. The formal specification contains detailed comments, too.

4.1 Needed Stuff for Forward Computation

Up to a parallel call is reached, the usual run-time stack frames are considered, i.e. choice points and environments. From this point, another frame must be used to hold the information due to the fork point, in particular the available parallel siblings and so on. Following the previous nomenclature, we will call these frames, *parcall* frames. In each parcall frame we keep the information needed for each parallel sibling in slots, for instance, the application name, its state (whether it is already computed, or it is waiting to be stolen), and the result of the computation (For this frame and the following data structures see Figure 2).

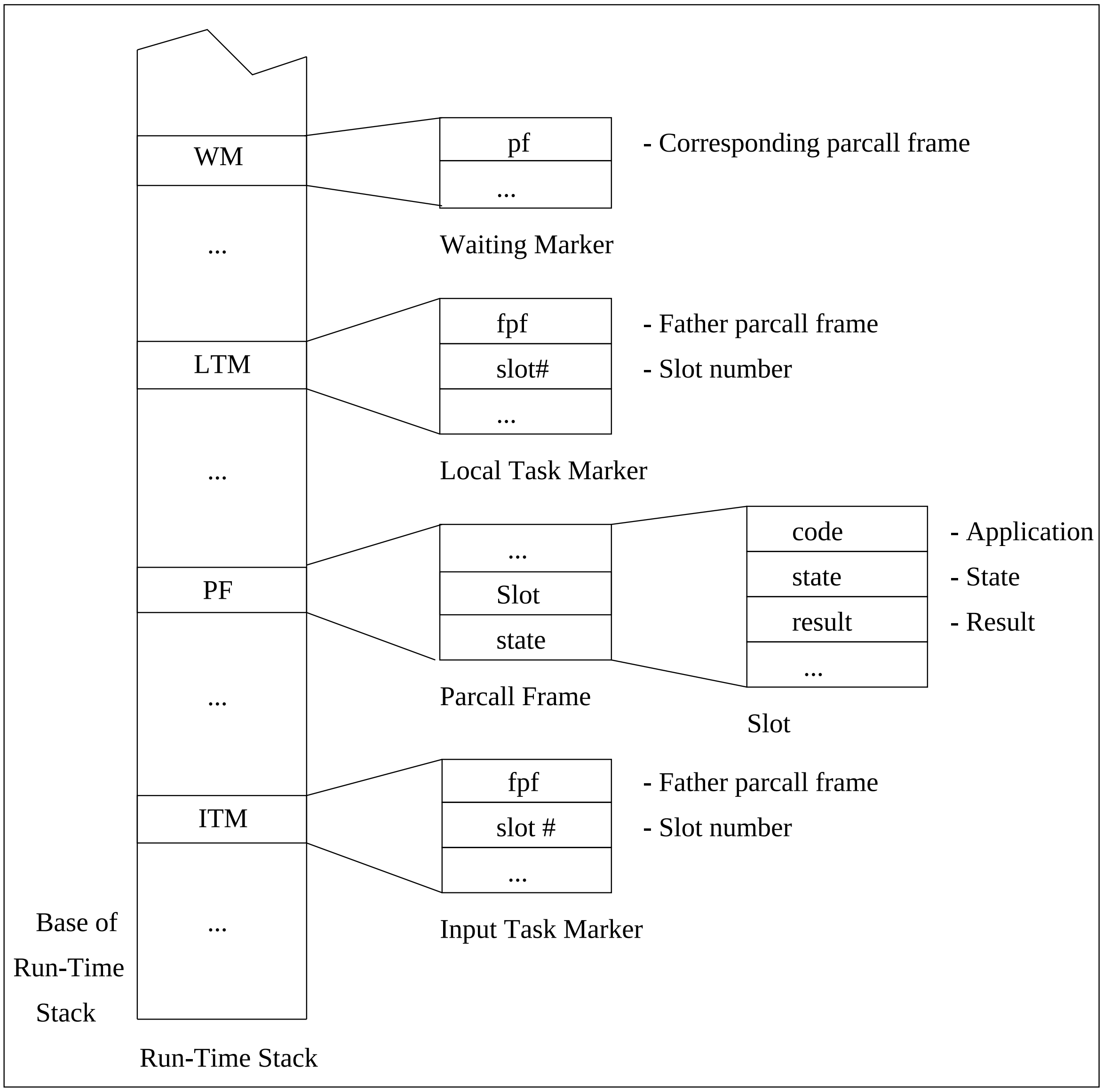

Figure 2: Needed frames to support parallelism

Since we allow the local computation of parallel siblings [6] given in the *active* parcall

[6]High workload may lead to circumstances under which there are no available resources to pick up the newly created task.

frame [7], we provide some mechanisms to isolate pieces of the stack belonging to different computations, which are local children of the parent machine. We do it by means of another frame, the so-called *local task marker* that keeps information to denote the parent parcall frame and the corresponding parallel sibling. Moreover, in order to keep computation in remote machines also isolated we will use a similar frame, the so-called *input task marker*, which contains similar information to the local task marker.

Parcall frames, local task markers, and input task markers constitute the new data structures needed to support parallelism in forward computation in the presented model. A slight description of what happens when a fork point is reached comes next.

In the local machine, a parcall frame is pushed onto the stack in order to notify other machines of the new available work. Furthermore, a local task marker is pushed to denote the local computation of the oldest application at that point. And then, the local computation is started whereas remote machines can pick up work from the recently created parcall frame. Processors will look for work among remote data areas in their own local run-time stacks and if they find a suitable sibling, they just pick it up by noting locally that the sibling is stolen by themselves. When the local computation is finished, the local machine informs the local parcall frame about the result. If some work still remains in the parcall frame, it can be locally picked up by pushing another local task marker and starting the new computation as before. If there is no more work available and the machine runs out of work then it simply waits until the completion of the remote parallel siblings.

In order to start the remote computation, an input task marker is pushed onto the run-time stack to denote the beginning of the new computation. Once the sibling is computed, the outcome is notified to the parent machine and the machine searches for new parallel siblings.

4.2 Extending Frames to Support Backward Computation

When a failure happens we must be able to recognize the point at which the failure occurs in order to behave correctly. We distinguish three states in which a failure can happen:

- A sequential computation runs out of alternatives provided that the next choice point is below the topmost join point (*outside* state). In order to detect this situation another frame is used: the so-called *wait marker* that holds among other information the parcall frame it corresponds to. It is pushed onto the run-time stack whenever a parallel call has been completed and the forward computation resumed, i. e. , when the join point has been successfully reached. Since we maintain the sequential order of delivered solutions, we must find the rightmost sibling to be asked for alternatives. This can be inferred by inspecting the slots in the parcall frame that correspond to the current topmost wait marker. Following the backtracking criteria presented in the previous section, a redo is sent to the proper machine. The failed sibling might be remotely or locally computed. If there are pending alternatives, then the failure operation is simply done in the same way as in the WAM, by inspecting the topmost choice point looking for new alternatives. If it has no pending alternatives for the computation, then the state of the remote machine must be restored.

[7] The active parcall frame of a given machine is the newer parcall frame with siblings not yet computed.

- A failure is computed during *inside* state for a parallel sibling (during forward computation). If necessary, the machines kills the remote ones and waits for the kill acknowledgements before switching to backward mode. This is a particular case of *intelligent* backtracking [10], since one sibling child is known to definitely deliver only failure. The kill procedure must restore the machine state before the corresponding input task marker was pushed. The restoration of the previous state is efficiently done by restoring the corresponding registers, and by resetting the variables annotated in the trail, as in sequential systems.

- Finally, we consider the case when a failure is computed for a local (resp. remote) parallel sibling, provided that the next choice point is below the topmost local (resp. input) task marker and the state is *outside*. Unlike inside state, we try a redo for the rightmost sibling b to the left of the failed one, killing the parallel siblings to the right of b. The parallel siblings to the right of b are killed. If the sibling b computed a non failed result, then the parallel forward computation for the siblings to the right of b is resumed. Otherwise, the failure procedure is applied again.

In all the aforementioned cases, it is necessary to kill remote or local parallel siblings that have been stolen because their outcomes are not needed anymore. At this point we reach another efficient behaviour of our model thanks to the preservation of the order, because the kill procedure utilizes the fact that the last incoming kill message corresponds to the topmost input task marker. The formal specification will present in detail how this behaviour is embodied in our model. But firstly, the specification language will be described in the following section.

4.3 The Specification Language

For the abstract formal specification we have chosen an imperative style of presentation instead of the more common functional style. But this description is comparably comprehensive. The functional style sometimes lacks readability, which is caused by its too compact form.

In our language, indentation determines the statements which belong to a given body, there is no use of the typical *begin-end* grouping symbols. Conditionals are expressed as usual. The operations are assumed to be executed in the sequential manner, although the parallel progress is applicable. But the sequential behaviour is more adequate due to its closer relation to the final VHDL specification. The description uses sequential assignments $\leftarrow$, Pascal like procedure calls, (informal) conditionals, and special primitives (e.g. $=$, $<$). The access to the components of structures or to elements of arrays is denoted with operators "." and "[ ]". The difference between pointers to objects and the objects themselves are sometimes neglected because of its small meaning at this abstract level of description. Parentheses are used to group arguments in function or procedure calls (e.g., *kill(pf, 1, pf.#slots)*), and also to refer to objects (e.g., *(itm.fpf).#slots* refers to the contents of the field *#slots* in the parcall frame annotated in the field *fpf* of the current input task marker *itm*).

Furthermore we extend the semantic with side effects, i.e. synchronization, and the syntax with informally described operations. Since there are some critical regions which may be updated concurrently (e.g. *kill*, *ack*, and *lock* fields), accesses to them are supposed to be managed in a semaphore scheme.

The specification merges precise descriptions with informal ones, that are distinguished by preceding signs + and ∘ for the precise and informal points, respectively. By this distinction we focus on the more important points and improve the readability of the whole algorithm. In particular, specifications which are not important about the presentation of the parallel behaviour and which can be consulted in other papers or are straightforwardly implemented are omitted (e.g. *continue with forward running mode*).

Together with the title of each piece of specification, its formal name is given in italics.

4.4 Formal Specification

In this section we describe the machine behaviour in terms of its parallel related actions. Since the topics concerning the narrowing machine are more widely known, we delay the presentation to the final VHDL specification, while we go deep inside the description of the parallel behaviour through its formal specification. Below, we mention several important issues of the parallel system.

Among the most representative topics of a parallel system we can find the failure operation, the scheduling policy, and the killing procedure. Nevertheless, topics related mainly with the forward computation like fork and join point arrangement will also be covered in this section.

The criteria in all the solutions is that parallelism will not only be exploited in terms of And parallelism, but also in all the duties related with the machine operation, like the aforementioned issues. For this purpose, idle machines will take charge of the duties which are allowed to be performed in parallel. This criteria will be reflected, for instance, in the description of the killing procedure or in the scheduling policy. As usually done in parallel schemes, some kind of synchronization must be performed between the parallel machines. In our case this will be done by performing an efficient lock mechanism of parcall frames. Let us analyze some representative issues.

- **Locking parcall frames**.
  Parcall frames are locked by the local or remote machines which are analyzing the information held in the parcall frame searching for definitory values that may conclude a given computation. This exploration is performed when finishing the computation of a parallel sibling. The lock of the parcall frame avoids any update by other machines. This is implemented using a field in each parcall frame that denotes its locked or unlocked state.

- **Finishing a given computation**.
  When a remote or the local machine [8] has finished its computation yielding a successful or a failed result, it tries to update the corresponding parcall frame with the proper information (the outcome, the presence of alternatives, the failed result, ...). If the parcall frame is locked, then the machine must wait until one of the following happens:

  - a kill notification is received, or
  - the parcall frame becomes unlocked.

[8] From now on, remote machine stands for the machine which has stolen a parallel sibling, whereas local machine stands for the machine that has just pick up a sibling from its own 'task queue'.

These are the two unique events that may happen in this scheme. The first one because a brother or an ancestor deliver a kill notification, and the second one because the computation of a sibling has finished with the updating of the parcall frame. When the parcall frame can be accessed in absence of a kill notification, then the machine can lock the parcall frame and update the corresponding entries. It is important to note here that each machine finishing the computation of a parallel sibling will explore the parcall frame and take charge of the proper actions, instead of disturbing the local machine and therefore improving resource handling.

- **Sending kill notifications**.
  Kill notifications may be generated in presence of failure or a definitory result. When considering inside mode, a failure received for a given sibling implies the killing of the parallel siblings. In outside mode and in presence of failure, all the parallel siblings to the right of the rightmost one with alternatives are also killed. When a definitory value has been obtained so that the siblings to the left have returned non-definitory values, then the siblings to the right are killed.

  Thanks to the order kept in the system, we are sure that the incoming kill notifications correspond to the topmost input marker in the run-time stack as we mentioned above. This allows the design of kill and acknowledge procedures to perform the needed synchronization that can be done by a counter holding the number of received kill notifications. Since the information regarding the corresponding sibling is held in the input task marker, no further information has been included in the kill notification. The target machine will perform the appropriate actions such as restoring the machine state, the recursively application of the killing procedure, and maybe the transmission of new kill notifications.

- **Looking for work**.
  As stated above, in order to prevent the garbage slot and the trapped goal problems refereed in [10], we put restrictions on the parallel siblings that a machine can steal from the network. The key problem is whether the sections can be pushed on top of the stack so that they will be recovered first on backtracking:

  - If the machine is free (i.e., with an empty run-time stack), it is allowed to steal any available sibling.
  - If the machine has a computed sibling t on top of its stack (i.e. this machine has reached the join point of the parcall frame corresponding to the topmost input task marker), then if the parent parallel call is active (there are siblings not yet computed), then it is allowed to steal sibling to the right of t (i.e., those siblings belonging to the same parcall frame. See Figure 3-a). Tasks to the left of t are not allowed to be stolen since they are known to be newer than t in the backtracking order.
  - Finally, a machine is not allowed to steal any work from another ancestor machine unless the join point of the ancestor machine has been reached by the computational path which p belongs to (See Figure 3-c).

  We summarize that a machine p is able to steal work from a given fork point if all the computational paths in which p takes part have reached the corresponding join points. This means that not only the immediate children of the considered fork

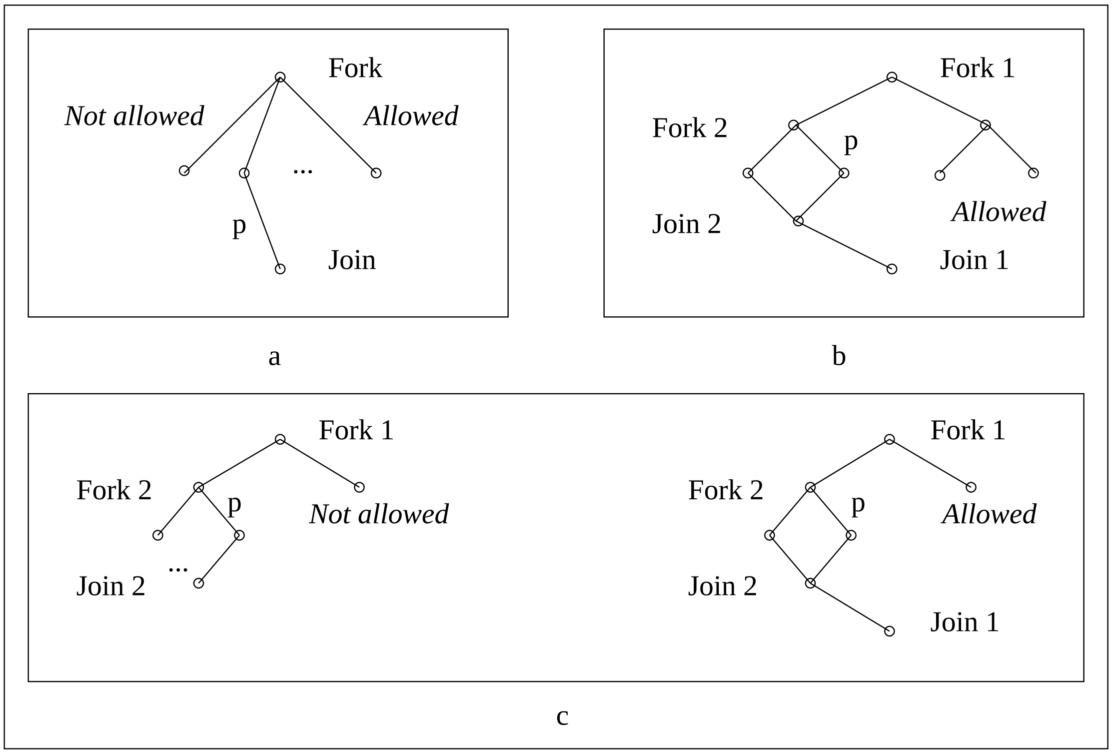

Figure 3: Stealable siblings

point, but also any descendant can be stolen (See Figure 3-b). Even when these restrictions seem to be run-time wasting, we will see in the formal specification how it can be efficiently done.

As previously mentioned, the local machine with an active parcall frame is restricted to steal work only from its active parcall frame. On the other hand, a remote processor which has finished its last computation will firstly look on its parent parcall frame, known from the topmost input task marker. If all the siblings have been computed and the parent join point has been reached there is no pending work in the parent parcall frame. In this case the next ancestor parcall frame is consulted for available work.

A given parcall frame may definitely have available work if there is at least a sibling ready to the left of the leftmost definitory sibling, or it may have work in its children because there are running parallel siblings.

When the leftmost stealable sibling is a ready sibling, the decision of what sibling to compute is clear, since this is the oldest available sibling. It is important to note that we are always looking for oldest siblings because in this way we are able to steal more work because of the precedence condition. We may consider that if a machine p steals a sibling to the right of the ready sibling t, then p will not be able to steal t in the current forward computation anymore. When the leftmost stealable sibling t is a running sibling and there are other ready siblings to the right of t the decision is not so clear. At this point we may follow a lazy strategy and steal the leftmost ready sibling. On the other hand, we may follow an eager strategy and fetch the leftmost running sibling for available work. In order to be able of stealing as much work as possible, and also to avoid speculative work for rightmost siblings (some running siblings to the left may deliver definitory results) we select the eager strategy. This strategy implies the suspension of the procedure when a locked parcall frame *pf* is fetched. After *pf* is unlocked, three cases are possible: the parcall frame becomes closed (non active), it has pending available work, or it is killed. The first two cases pose no problems since the procedure works in the same way, but the last one, if we consider that incoming kill messages may arrive when the look for work procedure has been invoked, must be carefully observed. It is clear that whenever a kill message arrives, this procedure must be stopped. If the incoming kill message arrives at the fetched stack in other processor, then we may consider the possibility of following the chain of parcall frames in reverse order looking for other available sibling. But this is a quiet hard duty because: first, we must prevent the interference of other killing messages when following the chain, and second, if we decide to store the parcall frames being fetched instead of following the chain, we must pay the extra overhead in memory consumption and complexity of the strategy. Both issues may suggest that the simplest solution may be the best, i.e., to simply start again the look for work procedure from scratch.

- **Receiving redo or kill notifications**.
  A redo notification consists only of the setting of a processor register. If a redo notification arrives, then an alternative computation is looked up by means of the topmost choice point in top of the stack or the corresponding parallel frames (this will be discussed in detail when the failure operation will be presented). If a kill notification arrives, then the computation up to the topmost input task marker is discarded. In the meantime, more kill notifications may arrive and therefore the previous states of the machine will be restored as often as kill notification are incoming. This procedure is done in parallel, and besides, there is no need for waiting for the acknowledges in the sequential order of the remote siblings, so that the speed of this procedure is also incremented in this way.

We will present a detailed specification in which low-level details have been taken into account, but before, we will present some features of the intended target machine.

Several decisions can be taken in order to denote and identify the frames in the run-time stack. We may consider, for instance, an explicit denoting of frames, marking the top of the stack with a pointer (maybe implemented with a register). In such a scheme, all the frames can be identified by means of the top of the stack and following a reference chain linking the frames. However, the price paid in following the chain by the procedures which need to reach some frame, together with the price paid by the extra memory entry needed to denote the frame, seem to be not worthwhile if we compare it with a register-based scheme. We adopt the usual technique of maintaining pointers to topmost or active frames in the stack, so that the more often needed frames can be quickly reached. Of course, whenever a frame is pushed onto the stack, the corresponding pointer is updated. Below, the pointers we use are listed.

- *pf*: *Active* parcall frame. It denotes the parcall frame which the current working task belongs to. In general, it is not the topmost parcall frame in the stack.

- *itm*: Topmost input task marker.

- *ltm*: Topmost local task marker.

- *wm*: Topmost wait marker.
- *b*: Topmost choice point.
- *bopf*: Bottommost parcall frame.

The above pointers are closely related with the parallel actions. We also consider the WAM-like pointers h (top of the heap), t (top of the trail), e (topmost environment), and d pointing to the top of the data stack in which arguments and results are held.

There are some registers in the processors which are intended to store needed information for parallelism support, for instance:

- *#kill*: A field containing the number of input task markers to be discarded (killed).
- *#ack*: It contains the number of expected acknowledges in response to kill notifications, added to support the needed synchronization.

These fields together with the special flag *lock* in each parcall frame that are handled as semaphores indicating critical regions.

Next, we will present the formal specification of the parallel-related machine operations together with comments. We will review the actions taken when a successful outcome has been computed, the failure operation, the kill and redo notifications, the kill procedure, and the looking for work procedure.

- **Successful outcome**
  Below, the specification for both the locally computed successful outcome and the remotely computed successful outcome are depicted.
  Before trying to update the parent parcall frame, the kill field is consulted to see whether the current computation has not been cancelled. This is accomplished by the procedure *lock parcall frame*, which returns a flag indicating whether a kill notification is present or not (see below its specification). If any kill notification is present, then the kill procedure (whose specification will be presented later) is activated. Otherwise, the result is updated in the corresponding slot frame (known by the field slot# in the topmost input task marker or local task marker). If a choice point has been pushed onto the corresponding run-time stack, then the pointer b is greater than the pointer *ltm* for a locally computed sibling, or the pointer *itm* for a remotely computed sibling. This means that another solution may be found for the current computation. The field *state* is updated with *alt* (standing for pending alternatives) or with *noalt* (standing for no pending alternatives). The field *definitory* is also updated with *true* if the computed result conforms a definitory argument position, and otherwise with *false*. If a definitory value is computed, then the siblings to the right are killed, waiting for the kill acknowledgement. If the siblings to the left have been computed, then the active parcall frame is safely closed (reaching join point) with a wait marker and the forward computation is resumed. Another possibility is that the current computation does not deliver a definitory value, and it is the last computed one so that a definitory argument to its right has been computed previously. In this case, the parcall frame is also closed. Note that each processor having delivered the successful result is allowed to perform the closure operations on the parcall frame and to resume the forward computation. If the local processor computes a definitory value and not all the siblings to its

left have been computed, then it remains idle, since the for workload strategy (see below) ensures that there are no more local siblings to pick up.

Specification for the Locally Computed Successful Outcome

+ lock parcall frame (kill)
+ If kill then
    + turn to kill running mode
+ else
    + pf.slot[ltm.slot#].result ← (top of the data stack)
    + pf.slot[ltm.slot#].state ← if(b > ltm, alt, noalt)
    + If definitory(pf.function, (top of data stack)) or pf.#slots = ltm.slot# then
        + pf.slot[ltm.slot#].definitory ← true
        + kill(pf, 1+ltm.slot#, pf.#slots)
        + wait
        + If all the siblings to the left of ltm.slot# have state ∈ {alt, noalt} then
            + push a wait marker
            + pf.bm ← outside
            + ltm ← pf.ltm
            + pf ← pf.fpf
            + unlock parcall frame
            ∘ continue with forward running mode
        + else
            + unlock parcall frame
            + get idle
    + else
            + pf.slot[ltm.slot#].definitory ← false
            + If exists a definitory sibling t_d to the right of ltm.slot# so that the siblings ltm.slot# + 1 ... t_d - 1 have been computed then
                + kill(pf, t_d + 1, pf.#slots)
                + wait
                + push a wait marker
                + pf.bm ← outside
                + ltm ← pf.ltm
                + pf ← pf.fpf
                + unlock parcall frame
                ∘ continue with forward running mode
            + else
                + look for local work

Specification for the Remotely Computed Successful Outcome

+ lock parent parcall frame itm.fpf (kill)
+ If kill then
    + turn to kill running mode
+ else

+ (itm.fpf).slot[itm.slot#].result ← (top of the data stack)
+ (itm.fpf).slot[itm.slot#].state ← if(b > itm, alt, noalt)
+ If definitory((itm.fpf).function, (top of data stack)) or (itm.fpf).#slots = itm.slot# then
    + (itm.fpf).slot[itm.slot#].definitory ← true
    + kill(itm.fpf, 1+itm.slot#, (itm.fpf).#slots)
    + wait
    + If all the siblings to the left of itm.slot# have state ∈ {alt, noalt} then
        + push (top of the data stack) onto parent's data stack
        + push a wait marker onto parent's run-time stack
        + (itm.fpf).bm ← outside
        + wait
        + (itm.procid).ltm ← (itm.fpf).ltm
        + (itm.procid).pf ← (itm.fpf).fpf
        + unlock parent parcall frame itm.fpf
        ∘ continue with forward running mode at parent processor
        + look for work
        + else
            + unlock parent parcall frame itm.fpf
            + look for work
+ else
        + (itm.fpf).slot[itm.slot#].definitory ← false
        + If exists a definitory sibling t_d to the right of itm.slot# so that
            the siblings itm.slot# + 1 … t_d - 1 have been computed then
            + kill(itm.fpf, t_d + 1, (itm.fpf).#slots)
            + wait
            + push a wait marker onto parent's run-time stack
            + (itm.fpf).bm ← outside
            + (itm.procid).ltm ← (itm.fpf).ltm
            + (itm.procid).pf ← ((itm.procid).pf).fpf
            + unlock parent parcall frame itm.fpf
            ∘ continue with forward running mode at parent processor
        + else
            + unlock parent parcall frame itm.fpf
            + look for work

- **Failure**
  Below, the specification for the first step of the failure operation is given. If the topmost choice point is below the topmost input task marker no more alternatives for the topmost computation exist on the stack. Therefore, the failure must be notified to the remote father process. If the topmost choice point is below the topmost local task marker, the failure corresponds to the local computation, and the failure must be propagated to the local father. If the topmost choice point is below the topmost wait marker, we enter the topmost (inactive) parcall frame to backtrack. Of course, this parcall frame is in outside state.

Specification for the Failure Operation

+ If b < itm then
  + remote failure
+ else If b < ltm then
  + local failure
+ else If b < wm then
  + outside failure
+ else
  ∘ sequential failure

In both remote and local failure we distinguish both inside and outside states of the parcall frame.

Specification for the Remote Failure Operation

+ If (itm.procid).(itm.fpf).bm = inside then
  + remote inside failure
+ else
  + remote outside failure

Specification for the Local Failure Operation

+ If procid.bm = inside then
  + local inside failure
+ else
  + local outside failure

When the failure happens in inside state, the field *state* in the corresponding slot is updated with the label *failed*. Afterwards, the remaining siblings of the parcall frame are killed, waiting for their acknowledgements. If the failure refers to a remote computation the state of the remote machine is restored from the topmost input task marker, and the machine looks for further work. If the failure refers to a local computation the (sequential) backward computation is started if all the sibling to the left are computed (all of them computed with non definitory results). If not all of them have been computed, then the machine becomes idle, since it is not worthwhile to steal more local siblings. Note that when a kill notification reaches the remote machine it behaves similar to a computed remote failure. Furthermore it decreases the number of kill notifications, the parent acknowledge field, and it restores the pointer to the input task marker. This also holds for the remote failure in outside state.

Specification for the Local Failure Operation in Inside Mode. *local inside failure*

+ lock parcall frame (kill)
+ If kill then
    + turn to kill running mode
+ else
    + pf.slot[ltm.slot#].state ← failed
    + If all the siblings to the left of ltm.slot# have been computed then
        + kill(pf, 1, pf.#slots)
        + wait
        + turn to backward running mode
    + else
        + kill(pf, ltm.slot# + 1, pf.#slots)
        + wait
        + unlock parcall frame
        + get idle

Specification for the Remote Failure Operation in Inside Mode. *remote inside failure*

+ lock parent parcall frame itm.fpf (kill)
+ If kill then
    + (itm.fpf).slot[itm.slot#].state ← killed
    + #kill ← #kill - 1
    + (itm.procid).#ack ← (itm.procid).#ack - 1
    + restore state itm
    + kill
+ else
    + (itm.fpf).slot[itm.slot#].state ← failed
    + If all the siblings to the left of itm.slot# have been computed then
        + kill(itm.fpf, 1, (itm.fpf).#slots)
        + wait
        + turn parent to backward running mode
    + else
        + (itm.fpf).slot[itm.slot#].state ← killed
        + kill(itm.fpf, itm.slot# + 1, (itm.fpf).#slots)
        + wait
        + unlock parent parcall frame (itm.fpf)
        + restore state itm
        + look for work

When the failure affects to a parcall frame in outside state, then an appropriate child is searched to be redone. This child corresponds to the rightmost sibling with alternatives so that there is no definitory siblings to its left. If such a sibling is not present all the siblings are killed, and the synchronization is necessary before turning to the backward running mode. If the sibling was locally computed a (sequential) local failure operation is performed. Otherwise, the remote machine is asked for further alternatives. Note that several operations are performed in parallel, for instance, the siblings are killed in parallel to the alternative computation on the remote machine. Once the remote machine delivers a result, three cases are distinguished. The first case is when the result is a failure. Then, the *local outside failure* procedure is recursively applied. The second case is when a definitory value is computed for the remote parallel sibling. This corresponds to the closing of the parcall frame, therefore resuming the forward running mode. Finally, the third case is when a non-failed, non-definitory result is computed. Then, the forward running mode is resumed in the local machine.

When searching for a suitable sibling to be backtracked from the remote machine, two possible situations may happen. If no sibling is found, then all the siblings must be killed. The remotely computed is noted as killed, the others are notified to be killed and a wait is performed. If the sibling is found, then is asked for a redo, and if it was also computed by the same remote process, then it is turned to backward mode, otherwise it continues to look for new work.

Specification for the Local Failure Operation in Outside Mode. *local outside failure*

+ pf ← wm.pf
+ lock parcall frame (kill)
+ If kill then
    + turn to kill running mode
+ else
    + t_d ← leftmost definitory sibling(pf, pf.#slots)
    + t_a ← rightmost alternative sibling(pf, t_d)
    + If $t_a = 0$ then
        + kill(pf, $t_a + 1$, pf.#slots)
        + wm ← pf.wm
        + wait
        + turn to backward running mode
    + else
        + kill(pf, $t_a + 1$, pf.#slots)
        + If t_a was locally computed then
            + wm ← pf.wm
            + wait
            + turn to backward running mode
        + else
            + pf.slot[t_a].state ← running
            + pf.slot[t_a].procid.redo ← on
            + wait
            + Wait until pf.slot[t_a].state $\neq$ running
            + If pf.slot[t_a].state = failed then
                + local outside failure
            + else
                + If definitory(pf.function, pf.slot[t_a].result) then
                    + push a wait marker
                    + pf.bm ← outside
                    + ltm ← pf.ltm
                    + pf ← pf.fpf

+ unlock parcall frame
∘ continue with forward running mode
+ else
∘ continue with forward running mode

Specification for the Remote Failure Operation in Outside Mode. *remote outside failure*

+ lock parent parcall frame itm.fpf (kill)
+ If kill then
  + (itm.fpf).slot[itm.slot#].state ← killed
  + #kill ← #kill - 1
  + (itm.procid).#ack ← (itm.procid).#ack - 1
  + restore state itm
  + kill
+ else
  + (itm.fpf).slot[itm.slot#].state ← failed
  + t_d ← leftmost definitory sibling(itm.fpf, itm.slot# - 1)
  + t_a ← rightmost alternative sibling(itm.fpf, t_d)
  + If $t_a = 0$ then
    + (itm.fpf).slot[itm.slot#].state ← killed
    + kill(itm.fpf, 1, (itm.fpf).#slots)
    + wait
    + unlock parent parcall frame itm.fpf
    + turn parent to backward running mode
    + restore state itm
    + look for work
  + else
    + (itm.fpf).slot[itm.slot#].state ← running
    + kill(itm.fpf, $t_a + 1$, (itm.fpf).#slots)
    + (itm.fpf).slot[t_a].procid.redo ← on
    ∘ wait
    + unlock parent parcall frame itm.fpf
    + restore state itm
    + If (itm.fpf).slot[t_a].procid = local procid then
      + turn to backward running mode
    + else
      + look for work

Finally, we present a typical specification for the sequential failure. It distinguishes two cases. The first one is when the pointer b points to the bottom address, i.e., there are no pending choice points. The other case assumes that a choice point is allocated on top of the stack and then, a recovery procedure is started. The heap, data stack, trail and run-time stack are restored, and the trailed variables are reset.

Specification for the Sequential Failure. *sequential failure*

+ If b = bottom then
  ∘ notify about the general failure computed
+ else
  + d ← b.d
  + e ← b.e
  + h ← e.h
  + unwind(e.t)
  + t ← e.t
  + ic ← b.ba

- **Kill procedure**
  The first piece of specification related with the kill operation is the turn to kill mode (*kill*). In this state, the machine changes its state to *kill* by setting the field *rm* (running mode). Afterwards, it performs as many kill operations on the topmost input task markers as the field *#kill* indicates. Since the kill notification for remote machines consists only of increasing a field, the local kill operation is performed in parallel on the siblings of the parcall frame. The trailed variables are reset and a wait for the acknowledge of the kill notifications is performed.

  Specification for the Kill Procedure. *kill*

  + rm ← kill
  + While #kill > 0
    + While pf > itm
      + kill(pf, 1, pf.#slots)
      + pf ← pf.fpf
    + itm ← itm.itm
    + #kill ← #kill - 1
  + unwind(itm.t)
  + wait
  + (itm.procid).#ack ← (itm.procid).#ack - 1
  + restore state itm
  + rm ← looking
  + look for work

  The kill notification consists of sending the notification firstly to the remote processes and performing a local kill procedure on the local computation. This is depicted below.

  Specification for Sending Kill Notifications. *kill(p, i, j)*

  + kill remote(p, i, j)
  + kill local(p, i, j)

A kill remote notification is performed, as already mentioned, by simply increasing the corresponding counter *#kill* at the machine containing the sibling to be killed. A group of notifications is sent by providing the range of siblings to be killed, from the leftmost one to the rightmost one. Only the siblings already computed ($state \in \{alt, noalt\}$) or those being currently computed ($state = running$) are sent kill notifications. For synchronization purposes, the #ack counter is increased with respect to the expected number of acknowledgements.

Specification for Sending Remote Kill Notifications. *kill remote(p, i, j)*

+ For s = i to j
    + If p.slot[s].state $\in$ {running, alt, noalt} and
      p.slot[s].procid $\neq$ local procid then
        + (p.slot[s].procid).#kill $\leftarrow$ (p.slot[s].procid).#kill + 1
        + #ack $\leftarrow$ #ack + 1

The first incoming kill notification for a local machine corresponds to the topmost local task marker, and it is only necessary to reset the variables and restore the machine state at the local task marker point, instead of performing a recursive restoration as in the usual backtracking mechanism. A field named *kill* for each machine is used to identify when a local kill is being performed, in order to perform the restoration only once at the oldest local task marker point. In the specification below, a call to *kill local* is performed. It returns in the last argument the rightmost local task which has been killed. If no local task is killed, then no restoration is performed since the siblings being killed are remotely computed. Starting from the leftmost sibling (sibling number 1), since this is always locally computed, the restoration would be performed at that point.

Specification for Sending Local Kill Notifications. *kill local(p, i, j)*

+ If kill = on then
    + kill local slots(p, i, j, l)
+ else
    + kill $\leftarrow$ on
    + kill local slots(p, i, j, l)
    + If l > 1 and l < top then
        + unwind((p.slot[l].xtm).t)
        + restore state ltm (p.slot[l].xtm)
        + pf $\leftarrow$ p
    + wait
    + kill $\leftarrow$ off

Finally, the specification for *kill local slots* is given below. A kill procedure is recursively started for each child of the corresponding slot. This is accomplished whenever parcall frames are allocated as children (at local or remote processes) of the corresponding slot. Furthermore, it computes the rightmost sibling which has been successfully killed. This specification completes the specification of the kill procedure.

Specification for Killing Local Slots. *kill local slots(p, i, j)*

+ l ← top
+ For s = j downto i
    + If p.slot[s].state ∈ {running, alt, noalt} and
      p.slot[s].procid = local procid then
        + l ← s
        + If (p.slot[s].xtm).spf ≠ bottom then
            + kill((p.slot[s].xtm).spf, 1, (p.slot[s].xtm).spf.#slots)

- **Looking for work procedure**
  In this procedure we distinguish three cases. The first one is when the local machine looks for work. As stated before, the strategy in this case is quite simple: the local machine looks on its own active parcall frame for ready siblings, searching from left to right siblings. If no work is found, then the machine waits for the completion of remote siblings.

Specification for Look for Local Work. *look for local work*

+ t_d ← leftmost definitory sibling
+ For i = ltm.slot# + 1 to t_d - 1
    + If pf.slot[i].state = ready then
        + push a local task marker
        + ic ← pf.slot[i].code
        + pf.slot[i].xtm ← ltm
        + pf.slot[i].procid ← local machine identifier
        + pf.slot[i].state ← running
        + unlock parcall frame
        ∘ turn to forward computation

The second one refers to machines which are looking for work from scratch, i.e., after the initialization phase, machines with empty stacks start to look for work in the machine that narrows the query. Finally, the third case stands for a remote machine which has stolen and computed a sibling *t*. In this case, the goal is to explore the parent parcall frame starting from the sibling to the right of *t*. If a ready sibling is found, then it is just stolen. If a running sibling is found, then the next descendant parcall frame is recursively fetched. These two cases are embodied in the specification below, which corresponds to the look for work procedure for processes different from the seed process. If the run-time stack is empty then any sibling is stealable, otherwise the precedence condition is applied by looking for work on the ancestor parcall frame annotated in the input task marker. The pointer *bopf* is used to denote the bottommost parcall frame in order to be able of following the chain of parcall frames.

Specification for Look for Work procedure. *look for work*

+ If e = bottom then
    + Repeat
        + If 1.bopf $\neq$ bottom then
            + look for work(1, bopf, 1)
    + until 1.rm = stop
+ else
    look for work(itm.procid, itm.fpf, itm.slot# + 1)

This last specification uses recursive calls of *look for work* which is presented below. As usual, before trying to lock the denoted parcall frame, the possible incoming messages are tested. If any kill notifications are requested, the machine starts the kill procedure. Otherwise, the parcall frame is locked. The work is searched from the given slot up to the leftmost definitory one (if it exists, else to the rightmost). If the analyzed slot is in a ready state, it is just fetched, and the forward computation resumes. If the respective sibling s is running, the descendant parcall frame (if exists) is searched for more work by inspecting the field xtm in s.

Specification for the Look for Work procedure. *look for work(id, pf, slot)*

+ While #kill = 0 and pf $\neq$ bottom
    + lock parcall frame (kill)
    + If kill then
        + turn to kill running mode
    + else
        + $t_d \leftarrow$ leftmost definitory sibling for id, pf
        + For i = slot to t_d - 1
            + If pf.slot[i].state = ready then
                + push an input task marker for slot i
                + e $\leftarrow$ id.e
                + ic $\leftarrow$ pf.slot[i].code
                + pf.slot[i].xtm $\leftarrow$ itm
                + pf.slot[i].procid $\leftarrow$ local proc id
                + pf.slot[i].state $\leftarrow$ running
                + unlock parcall frame pf
                ◦ turn to forward computation
            + If pf.slot[i].state = running then
                + vpf $\leftarrow$ (slot[i].xtm).spf
                + vid $\leftarrow$ if(id $\neq$ local proc id, (slot[i].xtm).procid,

local proc id)

+ If vpf $\neq$ pf then

+ unlock parcall frame pf

+ look for work(vid, vpf, 1)

+ unlock parcall frame pf

- **Miscellaneous operations**

  To finish the specification, we end with some minor specifications about the missing points above.

  For instance, a parcall frame locking is performed by using the primitive wait, which is intended to handle semaphores. In particular, the semantics of this wait primitive is extended with the notification of a kill operation, i.e., when a kill is ordered to a given machine, the second argument of the wait function is set.

Specification for Locking a Parcall Frame. *lock parcall frame pf*

  + wait(pf.lock, kill)

  Unlocking of parcall frames is handled in the same way as a typical signal primitive operation.

Specification for Unlocking a Parcall Frame. *unlock parcall frame pf*

  + pf.lock $\leftarrow$ false

  Changing the machine state is specified as follows:

Specification for Turn to Forward Running Mode. *turn to forward running mode*

  + rm $\leftarrow$ forward
  + narrow

Specification for Turn to Backward Running Mode. *turn to backward running mode*

  + rm $\leftarrow$ backward
  + failure

Specification for Turn to Kill Running Mode. *turn to kill running mode*

  + rm $\leftarrow$ kill
  + kill

Specification for Turn to Idle Running Mode. *turn to idle running mode*

+ rm ← idle
+ idle

Several events may happen in an idle state: incoming kill messages, incoming redo messages, a turn to the forward running mode and a local failure initiated by a remote machine. This is specified below.

Specification for Idle State. *idle*

+ Repeat
    + Case
        #kill > 0:
        + kill
        redo = on:
        + failure
        rm = forward:
        ∘ narrowing
        rm = backward:
        + failure
+ until true

The wait operation is simply performed by inspecting when the field *#ack* becomes zero, since it holds the expected number of acknowledgements.

Specification for Waiting for Acknowledgements. *wait*

+ Repeat
+ until #ack = 0

The function that computes whether an argument position is definitory is specified as follows.

Specification for the Definitory Function. *definitory(function, value)*

+ Case
    function = conjunction:
    + return value = false
    function = disjunction:
    + return value = true
    otherwise:
    + return false

Finally, we present the restoration-related operations.

$\boxed{\text{Specification for State Restoration up to a LTM. } \textit{restore state ltm(ltm)}}$

+ t ← ltm.t
+ h ← ltm.h
+ b ← ltm.b
+ d ← ltm.d
+ e ← ltm.e
+ ltm ← ltm.ltm

$\boxed{\text{Specification for State Restoration up to an ITM. } \textit{restore state itm(itm)}}$

+ t ← itm.t
+ h ← itm.h
+ b ← itm.b
+ d ← itm.d
+ e ← itm.e
+ pf ← itm.pf
+ ltm ← itm.ltm
+ wm ← itm.wm
+ itm ← itm.itm

5 The Parallel System

In this section we give a detailed presentation of the parallel system. We start with the abstract machine which is described together with the instruction set and the translation scheme. We end with the description of the shared memory system.

The parallel system consists of a set of abstract machines connected to a shared memory, which serves all the data requests.

5.1 The Abstract Machine

Even though many of the topics covered in this section have been already introduced when they were needed, this section groups all of them as a reference in order to get an overall impression of the machine.

The abstract machine is a parallel narrowing unit which has been designed from the point of view of the previous work on the sequential abstract machine [17].

Figure 4 depicts the architecture of the parallel system by showing the registers and memory areas it is composed of.

We distinguish four different kind of registers:

- *Usual narrowing registers.* They are used in a typical implementation of a sequential stack-based implementation of a narrowing machine [17]. The register e is used to

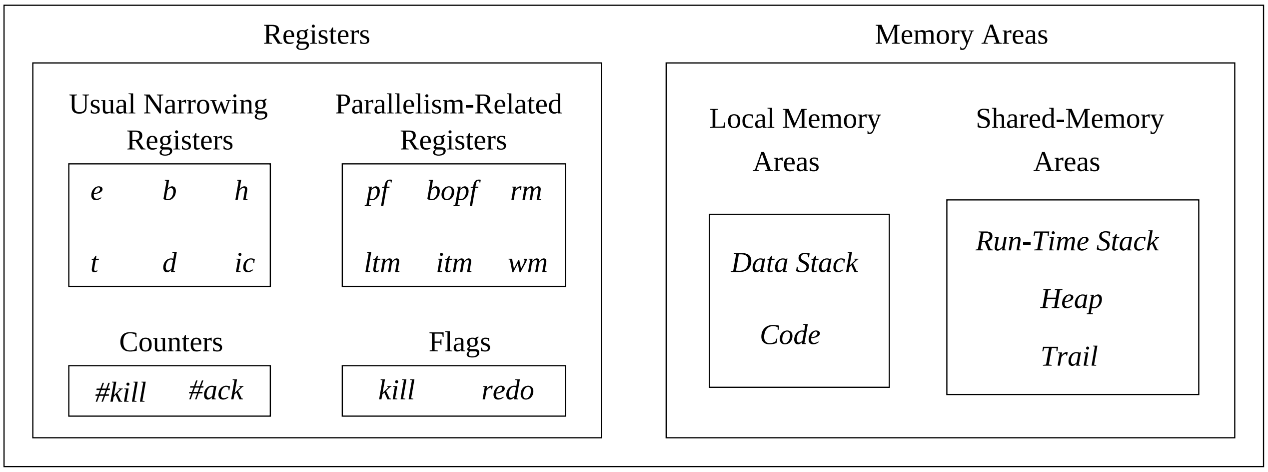

Figure 4: The Parallel System

denote the topmost environment, *b* is used for the topmost choice point (backtracking register), *h* points to the top of the heap, *t* points to the top of the trail, *d* points to the top of the data stack, and finally, *ic* points to the current instruction in the code area to be executed.

- *Parallelism-related registers.* They are used to deal with the pointing to the data frames needed for parallelism support. The register *pf* is used to denote the *active* parcall frame, while *bopf* points to the bottommost parcall frame. Registers *ltm*, *itm*, and *wm* are used to point to the topmost local task marker, to the topmost input task marker, and to the topmost wait marker, respectively. Finally, the register *rm* is intended to reflect the state of the machine (i.e., running, looking for work, performing a kill, or idle).

- *Counters.* We have two counters which are intended to manage the needed synchronization for kill operations. *#kill* holds the number n of incoming kill notifications corresponding to the n topmost input task markers (i.e., the n last computed computations). It is always increased by a remote machine. The register *#ack* stores the number of expected acknowledgements to the sent outcoming kill notifications. It is decreased by the remote machine as soon as the corresponding (remote) computation has been killed.

- *Flags.* The flag *kill* is set whenever a local kill procedure is being performed. It prevents the kill beyond the expected computation. The flag *redo* is set by a remote machine whenever a new solution is requested. The given machine is assumed to be in idle mode, and moreover, because of the precedence condition, the redo notification always refers to the topmost input task marker.

Figure 4 depicts the data areas placed in the memory, distinguishing those belonging to local and to shared memory areas. We review several memory data areas (except the registers), which are depicted together with their possible contents (data frames) in Figure 5. The structure of each data frame is shown in Figure 6.

- *Code.* It stores the abstract program to be executed. After initialization it becomes a read-only area.

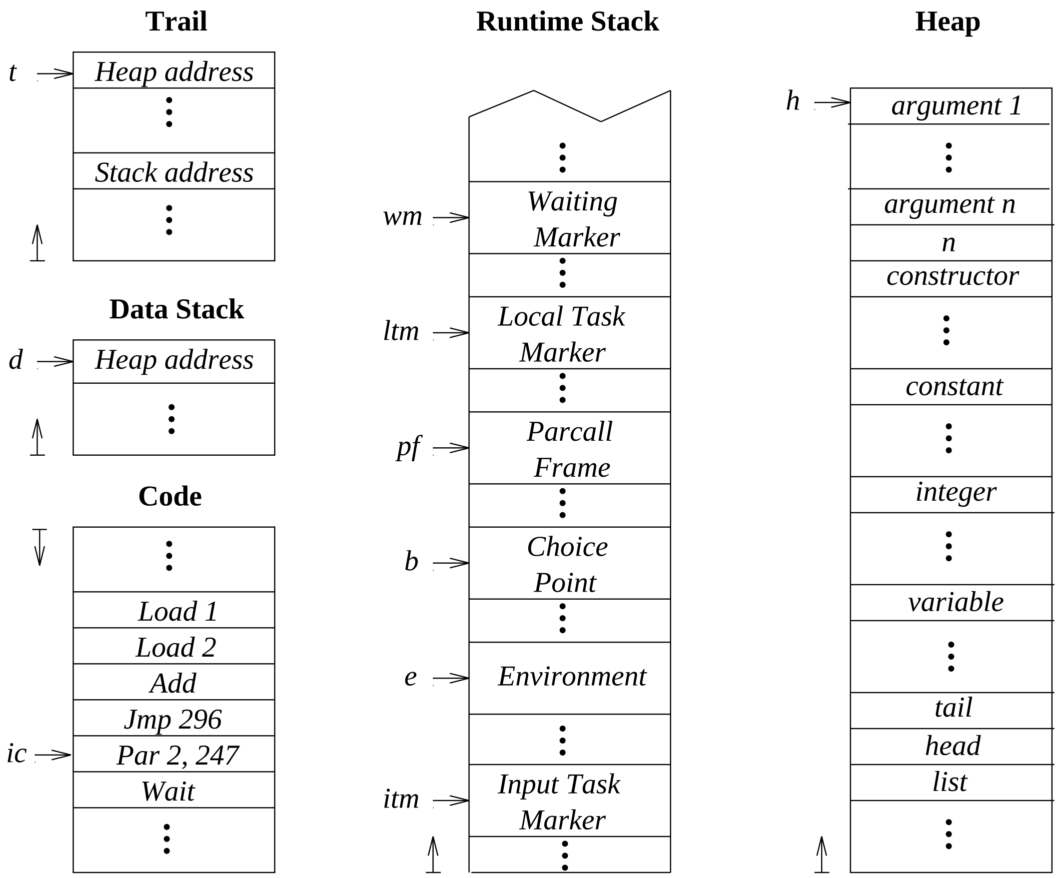

Figure 5: Some Data Areas of the Narrowing Engine

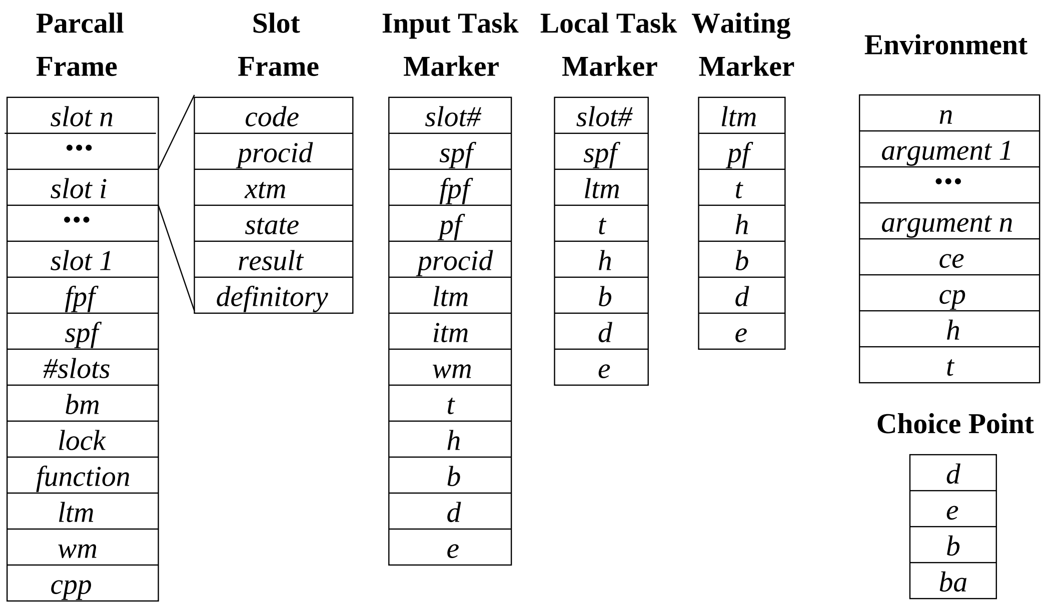

Figure 6: The Data Frames

- *Data Stack*. It holds both the call arguments to functions and the results returned by functions. The data stack of the machine which narrows the query will actually contain the result of the query. The contents of the data stack are pointers to the heap.

- *Runtime Stack*. It holds the control-related data frames to deal with forward and backward computation in both sequential and parallel execution.

  The usual frames in a stack-based machine are the choice points to control the selection of alternatives, and the environments which are used to control procedure calls. An environment has several fields to store the number of call arguments and the argument themselves (n, *argument 1*, ..., *argument n*), the continuation environment *ce*, the continuation point *cp* (i.e., the return address), and the saved heap h and trail t registers. h and t are placed at the environment instead of the choice point in order to take advantage of the dynamic detection of determinism [18]. The choice point stores the data stack register d, the environment register e, the choice point register b and the backtrack address *ba* which represents the next available rule (alternative) to be tried.

  The parcall frame identifies a fork point. It consists of the following fields:

  - *#slots* denotes the number of siblings in the parallel call. There are as many frames *slots* in the parcall frame as there are siblings. Each slot contains the information needed to know the computational state of the sibling, as well as the information for the machine which picks it up, i.e. the fields:
    * *code* which points to the piece of code which computes the corresponding sibling.
    * *procid* which holds the identifier of the machine computing the sibling.
    * *xtm* which points to the corresponding marker (a local task marker if it is locally computed, or an input task marker if it is remotely computed).
    * *state* which stores *ready* when a sibling is ready to be computed by any machine, *running* when the sibling is currently being computed, *alt* when the sibling is already computed and has pending alternatives which may be further tried, *noalt* stands for a computed sibling without pending alternatives, *failed* when the computation delivers failure, *killing* when it is being killed, and *killed* when the kill procedure actually ends.
    * *result* which points to the result in the heap.
    * *definitory*, a flag which denotes whether the sibling has been delivered with a definitory result.
  - *spf* and *fpf* are intended to hold the child parcall frame and the parent parcall frame, respectively. With the field *spf* the chain of parcall frames can be traversed from ancestors to descendants, which is needed when searching for available work. The field *fpf* is used in backward running mode when parcall frames are deallocated and the parent parcall frame is searched.
  - *bm* stores the backtracking mode: *inside* when the join point is not already reached, and *outside* when the join point has been reached.
  - *lock* is a flag denoting the locked or unlocked state of the parcall frame.

- *function* stores the kind of parallelism (*strict* or *non-strict*) to which the parcall frame belongs.
- *ltm* and *wm* store the local task marker and the wait marker previous to the parallel call. The registers *ltm* and *wm* are restored from these fields when the parcall frame is deallocated during backtracking.
- *cpp* is the continuation point after the join point. When the forward computation after the join point is resumed by updating *ic* with the content of *cpp*.

The local task marker delimits a local computation for a given slot in a parcall frame. It consists of several fields:

- *slot#* holds the identifier of the slot the computation belongs to.
- *spf* is the child parcall frame, i.e., the next parcall frame in the current computation.
- *ltm* is the previous local task marker. During backtracking, the register *ltm* is restored from this field.
- *t*, *h*, *b*, *d*, and *e*, are fields storing the respective registers. These registers are restored from these fields during backtracking.

The input task marker delimits a remote computation for a given slot in a parcall frame at the remote machine. It consists of several fields:

- *slot#* holds the identifier of the slot the computation belongs to.
- *spf* and *fpf* point to the child and to the parent parcall frame. They are used as before in following the parcall frame chain.
- *ltm*, *itm*, *wm*, *t*, *h*, *b*, *d*, and *e*, are fields storing the respective registers. The previous state is recovered during backtracking.

The wait marker indicates the join point. It consists of several fields:

- *pf* points to the parcall frame to which the wait marker belongs.
- *ltm*, *t*, *h*, *b*, *d*, and *e* store the current state of the machine.

• *Heap.* The heap contains all the data which are constructed by unification of terms or reduction of expressions. Entries may be variables, constants, lists[9], or data constructors.

• *Trail.* In this data area, pointers to the variables in the run-time stack and in the heap are *trailed* (i.e., the placement of the variables are annotated in order to be unwound whenever backward computation demands it).

5.1.1 The Instruction Set

In this section we give a more abstract description of the operational semantics. We distinguish several classes of machine instructions as follows:

[9]Due to the importance of lists we have explicitly distinguished them from generic data constructors, as well as we have also done with constants (arity-0 data constructors).

Graph instructions

- `Load` i loads the i-th local variable of the environment onto the data stack. This instruction ensures that the loaded address is *dereferenced*. Dereferencing means traversing the chain of pointers until either an unbound variable or a data term is reached.

- `StoreConstr` c n generates a new constructor node of arity n. The addresses of the n components are given on the data stack and will be replaced by the address of the constructor node.

- `StoreConst` c generates a new constant node.

- `StoreApp` f n creates an application node. This node contains the address of the function f as well as the n arguments of the partial application.

- `InitGuardVars` *free* $n$initializes the local variables *free* to *free+n-1* of the actual environment. It is used to generate the free variables in the occurring in the guard of a function rule.

Unification Instructions

- `UnifyVar` i moves the pointer on top of the data stack to the i-th local variable in the environment. The instruction is applied whenever the local variable is known to be unbound at compile-time. This is ensured by the left linearity of Babel rules.

- `UnifyConstr` c n tries to unify the (dereferenced) top element on the data stack with the n-ary constructor c. If the unification succeeds, the pointer is replaced by the components of the constructor term. If the pointer refers to an unbound variable, the constructor term is generated, the variable bound to it, and the components initialized with unbound variables. In case of an unification failure backtracking occurs.

- `CheckEq` l represents the general unification procedure which recursively performs unification in case of lists or constructors by jumping to the code address l. Furthermore it generates an environment on the runtime stack in order to control the recursion.

Forward Control Instructions

- `Call` f performs a subroutine (sequential) call. A new environment is allocated on top of the runtime stack with the corresponding arguments taken from the data stack. Further needed information about the function can be found in the symbol table denoted by the reference f.

- `Proceed` f is equivalent to Call, but with the optimized handling of tail recursion.

- `Apply` n applies an application node to the n arguments above it on the data stack. If the argument list is complete a new environment is generated and the execution proceeds with the execution of the referring function, otherwise the argument list will be inserted into a copy of the application node.

- `Return` successfully finishes a function call by setting the instruction pointer to the return address and restoring the environment. Furthermore, the old environment is deleted provided that it is located on top of the stack.

- `Jump` l, `JumpIfTrue` l, `JumpIfFalse` l perform simple and conditional jumps. The conditional jump instructions request the top of the data stack for the Boolean value and pop this entry.

Backward Control Instruction The first six instructions correspond to the same commands and the choice point mechanism in the Warren Abstract Machine.

- `TryMeElse` l records the current state of the machine by creating a choice point on top of the stack. The choice point contains all the information necessary to recover this state when backtracking to the next alternative at program address l. The execution continues with the next instruction.

- `RetryMeElse` l restores the state from the choice point, resets the backtrack address of the choice point to l, and continues with the next instruction.

- `TrustMe` restores the state from the choice point and deallocates the choice point on top of the stack. The last alternative will be tried subsequently by continuing with the next instruction.

- `Fail` immediately leads to backtracking.

- `DynamicCut` tests whether any global variable bindings have not been done when executing the code for the current function. If the test is successful, this rule is the only one which remains applicable according to the non ambiguity restriction of Babel. Then, newer environments and choice points on top of the stack can be safely removed.

Process Instructions

- `Par` n l creates a parallel call frame with n slots and initializes all necessary components. Finally, a jump to the wait address l is performed.

- `Wait` n waits for the termination of the parallel children of the current parallel call frame. This represents the join point of the parallel computation. If the parallel call frame still has got available work, then the machine is allowed to steal this work for local execution.

- `WaitTab` f dd n denotes a table for each sibling of the parallel call, containing the address f of the code which narrows the sibling n, and the maximum data stack usage dd.

- `ConjWait and DisjWait` take advantage of the Boolean primitives and are adjusted to the parallel execution of non-strict conjunctions and disjunctions.

- `SendResult` n finishes a successful computation of the n-th parallel son. The computed result is located on top of the data stack and will be transferred to the parent machine.

- `SendFail` n notifies that no successful result has been computed.
- `Idle` is executed if the machine is in idle mode and forces the machine to demand further work.

Built-ins

- `Add, Subtract, Multiply, Divide`, and `Modulo` are the binary arithmetic primitives, which perform the corresponding operation on the arguments taken from the data stack.
- `GreaterThan, GreaterOrEqual, LessThan`, and `LessOrEqual` are the comparative primitives which operate on two arguments taken from the data stack.
- `And, Or, Xor`, and `Not` are the bitwise logical primitives which take their operands from the data stack.

Other Commands

- `More` asks the user whether more solutions are to be searched.
- `Stop` stops the computation.
- `InitPrint, Print, PrintChar` c, are responsible for the output of the solutions.
- `JumpIfEol` l, `JumpIfList` l, and `JumpIfData` l perform conditional jumps and are only used for technical reasons in the output procedure.

5.1.2 The Translation Scheme

After having introduced the instruction set, this section is devoted to the translation of Babel programs into PBAM code (standing for Parallel Babel Abstract Machine) code. We will use the well-suited functional style for the description of the translation scheme, unlike the more imperative formal representation of the abstract machine, in which we need other mechanisms to express concurrency. Italics are used for the identifiers of translation schemes, while a normal font is used for machine instructions and arguments). The symbols %% are used to denote a comment or condition which must be met by the corresponding rule in which they occur.

This presentation starts with the *progtrans* scheme, which produces some initial code for setting the machine into the looking for work state (Idle instruction), to stop the machine (Stop instruction) and several set of instructions needed to handle output (*print_prelude*) and some primitives (through the schemes *equality_prelude*, and *ho_prelude*). Next, the translation of the functions is generated by the *functrans* scheme for each function in the program. The entry point in the code before starting the narrowing of the query is InitBam. The translation of the expression to be narrowed is given by the *exprtrans* scheme. Finally, a jump to the print section is given. The entry point to the print section (L_{Print}) is located in the *print_prelude* scheme.

progtrans:(Rule*)* $\rightarrow$Expr$\rightarrow$PBAMCode
progtrans($<<$f^j t$^j_{i,1}\ldots$ t$^j_{i,n^j}$:= e^j_i $\mid 1 \le i \le r^j >\mid 1 \le j \le p >$,e) ::=

	Idle	
L_{Stop}:	Stop	
	print_prelude	
	equality_prelude	
	ho_prelude	
	functrans($<$f^1 t$^1_{i,1}\ldots$ t$^1_{i,n^1}$:= e^1_i $\mid 1 \le i \le r^1 >$)	
	$\vdots$	
	functrans($<$f^p t$^p_{i,1}\ldots$ t$^p_{i,n^p}$:= e^p_i $\mid 1 \le i \le r^p >$)	
L_{Start}:	InitBam	lv$_e$ n dd L_{Print} L_{Stop}
	exprtrans(e, 0)	
	Jump	L_{Print}

Below, the *exprtrans* scheme is shown. Several rules for *exprtrans* define the code generated for each kind of expression: variables, constructors, applications, higher order functions, guarded expressions (*if then*, and *if then else*), primitives (conjunction, disjunction, equality, disequality, arithmetic operations (OP$_\oplus$), and negation), the *let-in* constructor, and finally the parallelism-related functions (*letpar-in*, the parallel conjunction, and the parallel disjunction).

The translation of a variable leads to the generation of the Load instruction. In case of a constructor or function application, the arguments are translated first. For the constructor, the constructor node is generated in the heap and its address pushed onto the data stack. In the case of a function a Call or Proceed instruction is generated depending on whether tail recursion can be applied. If not all the arguments are applied, an application node is generated by StoreApp, instead.

The translation of a higher order application is done by translating the expression and its arguments and finally generating an Apply instruction.

Guarded expressions are translated by generating the code for the guard, and then a conditional jump examining the value which is on top of the data stack after the evaluation of the guard. The translation of conditional expressions is similar. The same is valid for the sequential Boolean functions. The first argument of the function is translated, and the resulting value of the evaluation which is placed on top of the data stack is consulted in order to see whether the evaluation of the second argument can be skipped (i.e., when the value is definitory).

The equality expression is processed similar to the function application. Note that the equality prelude is called. The disequality is reduced to the equality expression by simply appending the negation instruction *Not*. The other primitive operations are handled by tranlating their arguments and performing the corresponding operation on it.

The *let-in* constructor which defines an environment for local variables is translated by generating each translation of the right hand side of the assignments, and next, if the left hand side is a variable, the UnifyVar instruction, which simply makes the local variable point to the value in the data stack. Finally, the translation scheme for the expression outside of the local environment is applied.

The parallel constructor *letpar-in*, which defines an environment for the local execution of the right hand side of each assignment, is translated as follows. The first instruction is a jump to the fork point. After that, the scheme *strictpartrans* is applied for each assignment. Next, the sequential version *let-in* of the *letpar-in* is found, which is accessed whenever the work load balancing strategy decides to transfer the control to the sequential part. The instruction Jump to the unification section comes next. The Par instruction denotes the fork point of n siblings, which is followed by the Wait instruction which waits until the completion of all the siblings. For each parallel sibling a WaitTab instructions is generated denoting the respective entry points L_{F_i} to the code. What follows is the unification code for the results. Finally, the rightmost expression is translated.

The translation schemes for the non-strict Boolean functions conjunction and disjunction are similar to the scheme for the *let-par*. The unification part and the last *exprtrans* occurrence are superfluous since DisjWait and ConjWait instructions take care of these duties.

exprtrans:Expr→ $\mathcal{N}$ →PBAMCode
exprtrans(X_i, tl) ::=
    Load   i
exprtrans(c $e_1 \ldots e_n$, tl) ::=
    exptrans(e_1,0)
    ⋮
    exptrans(e_n,0)
    StoreConstr   c n
exprtrans(f $e_1 \ldots e_n$, tl) ::=
    exptrans(e_1,0)
    ⋮
    exptrans(e_n,0)

$$\begin{cases} \text{Proceed} & \text{L}_\text{f}\ \text{n}\ \ \%\%\ \ \text{arity}_f = n, \text{tl} = 1 \\ \text{Call} & \text{L}_\text{f}\ \text{n}\ \ \%\%\ \ \text{arity}_f = n, \text{tl} = 0 \\ \text{StoreApp} & \text{L}_\text{f}\ \text{n}\ \ \%\%\ \ \text{arity}_f > n \end{cases}$$

exprtrans(e $e_1 \ldots e_n$, tl) ::=
    exprtrans(e, 0)
    exptrans(e_1,0)
    ⋮
    exptrans(e_n,0)
    Apply   n tl
exprtrans(if e_1 then e_2, tl) ::=
    exprtrans(e_1, 0)
    JumpIfFalse L_{Fail}
    exptrans(e_2, tl)
exprtrans(if e_1 then e_2 else e_3, tl) ::=
    exprtrans(e_1, 0)
    JumpIfFalse l_{false}
    exptrans(e_2, tl)
    Jump l_{cont}
l_{false}:   *exptrans*(e_3, tl)
l_{cont}:   ...
exprtrans(e_1,e_2, tl) ::=
    exprtrans(e_1, 0)

	JumpIfFalse	l_{false}
	exprtrans(e_2, tl)	
	Jump	l_{cont}
l_{false} :	StoreConst	false
l_{cont}:	...	

exprtrans(e_1;e_2,tl) ::=

	exprtrans(e_1, 0)	
	JumpIfTrue	l_{true}
	exprtrans(e_2, tl)	
	Jump	l_{cont}
l_{true}:	StoreConst	true
l_{cont}:	...	

exprtrans(e_1=e_2, tl) ::=

exprtrans(e_1, 0)
exprtrans(e_2, 0)

$$\begin{cases} \text{Proceed} & \mathrm{L_{Eq}}\ 2\ \ \%\%\ \ \mathrm{tl} = 1 \\ \text{Call} & \mathrm{L_{Eq}}\ 2\ \ \%\%\ \ \mathrm{tl} = 0 \end{cases}$$

exprtrans(e_1 ~=e_2,tl) ::=

exprtrans(~(e_1=e_2),tl)

exprtrans(e_1⊕e_2, tl) ::= ($\oplus \in \{+, -, *, /, \%, >, <, \leq, \geq\}$)

exprtrans(e_1, 0)
exprtrans(e_2, 0)
$\mathrm{OP}_{\oplus}$

exprtrans(~e, tl) ::=

exprtrans(e, 0)
Not

exprtrans(let t_1=e_1,...,t_n=e_n in e, tl) ::=

exptrans(e_1,0)

$$\begin{cases} \text{UnifyVar i} & \%\%\ \mathrm{t_1} \in \mathrm{Var} \\ \mathit{unifytrans}(\mathrm{t_1}, \varepsilon) & \%\%\ \text{otherwise} \end{cases}$$

⋮

exprtrans(e_n,0)

$$\begin{cases} \text{UnifyVar i} & \%\%\ \text{if } \mathrm{t}_n \in \mathrm{Var} \\ \mathit{unifytrans}(\mathrm{t}_n, \varepsilon) & \%\%\ \text{otherwise} \end{cases}$$

exprtrans(e,1)

exprtrans(letpar t_1=e_1,...,t_n=e_n in e, tl) ::=

	Jump l_{fork}	
l_1:	*strictpartrans*(e_1, t_1, 1, i_1)	
	⋮%% i_k=min$\{0, \mathrm{j} \mid \mathrm{X_j} \in \mathrm{var}(\mathrm{t_k})\}(1 \leq \mathrm{k} \leq \mathrm{n})$	
l_n:	*strictpartrans*(e_n, t_n, n, i_n)	
$l_{sequential}$:	*exprtrans*(let t_1=e_1,...,t_n=e_n in e, tl)	
	Jump l_{cont}	
l_{fork}:	Par	n $l_{sequential}$
	Wait	n
	WaitTab	l_1 1
	⋮	

	WaitTab	l_n n
	unifytrans(t_1, Load i_1) %% $i_1 > 0$	
	$\vdots$	
	unifytrans(t_n, Load i_n) %% $i_n > 0$	
	exprtrans(e,1)	
l_{cont}:	...	

exprtrans(e_1 && ... && e_n, tl) ::=

	Jump l_{fork}	
l_1:	*partrans*(e_1, 1)	
	$\vdots$	
l_n:	*partrans*(e_n, n)	
$l_{sequential}$:	*exprtrans*(e_1, ..., e_n, tl)	
	Jump l_{cont}	
l_{fork}:	Par	n $l_{sequential}$
	ConjWait	n
	WaitTab	l_1 1
	$\vdots$	
	WaitTab	l_n n
l_{cont}:	...	

exprtrans(e_1 \|\| ... \|\| e_n, tl) ::=

	Jump l_{fork}	
l_1:	*partrans*(e_1, 1)	
	$\vdots$	
l_n:	*partrans*(e_n, n)	
$l_{sequential}$:	*exprtrans*(e_1; ...; e_n, tl)	
	Jump l_{cont}	
l_{fork}:	Par	n $l_{sequential}$
	DisjWait	n
	WaitTab	l_1 1
	$\vdots$	
	WaitTab	l_n n
l_{cont}:	...	

The translation scheme *unifytrans* which is applies to terms. In case of a variable, the given loadinstr instruction on the i-th variable is the unique instruction. In case of a constructor, the loadinstr instruction comes first, then the UnifyConstr instruction, and the *unifytrans* scheme is recursively applied for each argument of the constructor.

unifytrans:Term→({Load}×$\mathcal{N} \cup \varepsilon$) →PBAMCode

unifytrans(X_i, loadinstr) ::=

	...	

unifytrans(c t_1... t_n, loadinstr) ::=

	loadinstr	
	UnifyConstr	c n nlv
	unifytrans(t_1, Load nlv)	

$\vdots$

unifytrans(t_n, Load nlv+n-1)

Below, the *partrans* and *strictpartrans* schemes are given. The first one consists of the TryMeElse instruction pointing to the fail label, next the translation *exprtrans* and the SendResult instruction, which sends the computed result for the n-th child to the parent.

partrans:Expr→ $\mathcal{N}$ →PBAMCode
partrans(e, n) ::=
TryMeElse l_{fail}
exprtrans(e, 0)
SendResult n
l_{fail}: SendFail n
strictpartrans:Expr→Term→ $\mathcal{N}^2$ →PBAMCode
strictpartrans(e, t, n, i) ::=
TryMeElse l_{fail}
exprtrans(e, 0)
$\left.\begin{array}{l}\textit{unifytrans(t, }\varepsilon\textit{)}\\ \text{StoreConst true}\end{array}\right\}$ %% i=0
SendResult n
l_{fail}: SendFail n

The *guardtrans* scheme for an expression *e* with and without dynamic cut optimization [10] is given below. Firstly, the InitGuardVars or the DynamicCut instructions are generated for the optimized and non optimized cases, respectively, followed by the translation of the expression given by *exprtrans*.

guardtrans:Expr→PBAMCode
guardtrans(e) ::=
InitGuardVars gv n %% Without dynamic cut optimization
exprtrans(e, 1)
guardtrans(e) ::=
DynamicCut %% With dynamic cut optimization
exprtrans(e,1)
guardtrans(if e_1 then e_2) ::=
InitGuardVars gv n
exprtrans(e_1, 0)
JumpIfFalse L_{Fail}
DynamicCut
exprtrans(e_2, 1)

The translation of a rule is guided by the *ruletrans* scheme, which starts with the *unifytrans* scheme for each argument of the left hand side of the rule. It continues with the translation of the right hand side of the rule through the above scheme *guardtrans* and ends with the Return instruction.

[10] See [18] for details.

ruletrans:Rule→PBAMCode

ruletrans(f $t_1 \ldots t_n$:= e) ::=

    unifytrans(t_1, Load 1+lv_f)
    ⋮
    unifytrans(t_n, Load n+lv_f)
    guardtrans(e)
    Return

Below, the *functrans* scheme which is applied to a (finite) set of rules is given. The first rule shows the translation for only one rule, while the second one deals with more than one rule. The first instruction *Tab* stores the symbolic information of each function and is used by the instructions *StoreApp, Call, Proceed*, and *Apply*.

functrans:Rule* →PBAMCode

functrans(<f $t_{1,1} \ldots t_{1,n}$:= e_1 >) ::=

L_f:	Tab	lv_f $arity_f$
	ruletrans(f $t_{1,1} \ldots t_{1,n}$:= e_1)	

functrans(<f $t_{i,1} \ldots t_{i,n}$:= e_i \| $1 \le i \le r$ >)::=

L_f:	Tab	lv_f $arity_f$
	TryMeElse	l_2
	ruletrans(f $t_{1,1} \ldots t_{1,n}$:= e_1)	
l_2:	RetryMeElse	l_3
	ruletrans(f $t_{2,1} \ldots t_{2,n}$:= e_2)	
	⋮	
l_r:	TrustMe	
	ruletrans(f $t_{r,1} \ldots t_{r,n}$:= e_r)	

The *print_prelude* scheme deals with the translation needed to support output when printing constructors and lists.

print_prelude ::=

l_{ploop}:	PrintChar	44 %% ','
$l_{pconstr}$:	Print	$l_{pconstr}$ l_{plist} l_{pccont}
	JumpIfFalse	L_{Stop}
l_{pccont}:	JumpIfData	l_{ploop} 1
	PrintChar	41 %% ')'
	StoreConst	true
	Return	
l_{plloop}:	PrintChar	44 %% ','
l_{plist}:	Print	$l_{pconstr}$ l_{plist} $l_{plcont1}$
	JumpIfFalse	L_{Stop}
$l_{plcont1}$:	Load	1
	JumpIfEoL	$l_{plcont2}$
	JumpIfList	l_{plloop}
	PrintChar	124 %% '|'
	Print	$l_{pconstr}$ l_{plist} $l_{plcont2}$

	JumpIfFalse	L_{Stop}
$\mathrm{l}_{plcont2}$:	PrintChar	93 %% ']'
	StoreConst	true
	Return	
L_{Print}:	InitPrint	
	Print	$\mathrm{l}_{pconstr}$ l_{plist} $\mathrm{l}_{ploopvar}$
	JumpIfFalse	L_{Stop}
$\mathrm{l}_{ploopvar}$:	PrintVar	l_{pexit}
	Print	$\mathrm{l}_{pconstr}$ l_{plist} $\mathrm{l}_{ploopvar}$
	JumpIfTrue	$\mathrm{l}_{ploopvar}$
l_{pexit}:	More	
	JumpIfFalse	L_{Stop}
L_{Fail}:	Fail	

The *equality_prelude* scheme which is applied to data constructors is given below.

equality_prelude ::=

L_{Eq}:	Load	1
	Load	2
	CheckEq	$\mathrm{l}_{eqconstr}$
	Return	
$\mathrm{l}_{eqconstr}$:	UnifyVar	1
	Load	1
	CheckEq	$\mathrm{l}_{eqconstr}$
	JumpIfFalse	l_{eqfail}
	JumpIfData	$\mathrm{l}_{eqconstr}$ 2
	StoreConst	true
	Return	
l_{eqfail}:	StoreConst	false
	Return	

Finally, the *ho_prelude* which shows the translation regarding the primitives in $\oplus$ is given. Firstly, the arguments are loaded onto the stack and then the corresponding operation is applied.

ho_prelude ::=

$\mathrm{l}_{\oplus}$:	Load	1 ($\oplus \in \{+, -, *, /, \%, >, <, \leq, \geq\}$)
	Load	2
	$\mathrm{OP}_{\oplus}$	
	Return	

Finally, we will show the code obtained by translating the following example program, the parallel version of the fibonacci program.

The following Babel program:

```
fib X := if
          (X>1)
        then
          letpar Y1 = fib (X-1),
                 Y2 = fib (X-2)
          in Y1+Y2
        else 1.
```

with the query fib(10), is translated as shown below, where italics are used for labels.

	Idle	
Stop	Stop	
PLoop	PrintChar	44
PConstr	Print	*PConstr PList PCCont*
	JumpIfFalse	*Stop*
PCCont	JumpIfData	*PLoop*
	PrintChar	41
	StoreConst	true
	Return	
PLLoop	PrintChar	44
PList	Print	*PConstr PList PLCont1*
	JumpIfFalse	*Stop*
PLCont1	JumpIfEoL	*PLCont2*
	JumpIfList	*PLLoop*
	PrintChar	124
	Print	*PConstr PList PLCont2*
	JumpIfFalse	*Stop*
PLCont2	PrintChar	93
	StoreConst	true
	Return	
Print	InitPrint	
	Print	*PConstr PList PLoopVar*
	JumpIfFalse	*Stop*
PLoopVar	PrintVar	*PExit*
	Print	*PConstr PList PLoopVar*
	JumpIfTrue	*PLoopVar*
PExit	More	
	JumpIfFalse	*Stop*
Fail	Fail	
Eq	Load	1
	Load	2
	CheckEq	*EqStruct*
	Return	
EqStruct	InitGuardVars	1 1
	UnifyVar	1
	Load	1
	CheckEq	*EqStruct*
	JumpIfFalse	*EqFail*

	JumpIfData	*EqStruct*
	StoreConst	true
	Return	
EqFail	StoreConst	false
	Return	
Add	Load	1
	Load	2
	Add	
	Return	
Subtract	Load	1
	Load	2
	Subtract	
	Return	
Multiply	Load	1
	Load	2
	Multiply	
	Return	
Divide	Load	1
	Load	2
	Divide	
	Return	
Module	Load	1
	Load	2
	Module	
	Return	
GreaterThan	Load	1
	Load	2
	GreaterThan	
	Return	
GreaterOrEqual	Load	1
	Load	2
	GreaterOrEqual	
	Return	
LessThan	Load	1
	Load	2
	LessThan	
	Return	
LessOrEqual	Load	1
	Load	2
	LessOrEqual	
	Return	
And	Load	1
	Load	2
	And	
	Return	
Or	Load	1
	Load	2

```
                Or
                Return
Xor             Load            1
                Load            2
                Xor
                Return
Not             Load            1
                Not
                Return
fib             InitGuardVars   1 2
                Load            3
                StoreNum        1
                GreaterThan
                JumpIfFalse     Label.1
                Jump            Label.2
Label.6         TryMeElse       Label.5
                Load            3
                StoreNum        1
                Subtract
                Call            fib
                SendResult      1
Label.5         SendFail        1
Label.8         TryMeElse       Label.7
                Load            3
                StoreNum        2
                Subtract
                Call            fib
                SendResult      2
Label.7         SendFail        2
Label.3         Load            3
                StoreNum        1
                Subtract
                Call            fib
                UnifyVar        1
                Load            3
                StoreNum        2
                Subtract
                Call            fib
                UnifyVar        2
                Load            1
                Load            2
                Add
                Jump            Label.4
Label.2         Par             2 Label.3
                Wait            2 3
                WaitTab         Label.6 2 1
                WaitTab         Label.8 2 2
```

```
Label.4     Load        1
            Load        2
            Add
            Return      3
Label.1     StoreNum    1
Label.0     Return      3
Start       InitBam     0 1 5 Print Stop
            StoreNum    10
            Call        fib
            Jump        Print
```

5.2 The Memory

Several memory models can be considered in designing a parallel model. The most representative ones are the shared and the distributed memory systems. In the shared memory model, the memory is shared by the processors in the network. The processors have direct access to the memory, but they must wait until the usage of the memory bus is allowed. In order to control the memory access, some bus protocol is needed. The access frequency is limited by the bandwidth of the memory. In these systems, the bottle-neck problem occurs when the number of processors in the network significantly grows. In the distributed memory model, each processor with local memory is typically connected to others through links according to some topology. There is no need for a bus protocol, but instead for a message passing mechanism which drives the messages through the links. In this case, the efficiency is bound by the maximum message flow which depends eventually on the topology considered and the message bandwidth. Systems like the Sequent machines are representatives of the shared memory model, while Transputer systems are examples of the distributed memory model.

Both memory models have some advantages and drawbacks that make them more suitable for different applications. When some processors in the network have to access common memory areas, the shared memory model seems to be more adequate, because the expensive message passing mechanism is avoided. If the memory areas are assigned to different processors the distributed memory system is more adequate, because the shared memory access protocol is not needed. If a small number of processors is considered, the shared memory model seems to be also worthwhile.

In the sequel we will focus our attention on a shared memory model in which several processors are connected to a common memory through the system bus which will be controlled by an access protocol. Nevertheless, some improvements can be embodied in this model. For instance, data areas that are known to be only locally accessed will be kept in local memory. Therefore, we propose a system in which common data areas are located in the shared memory, while the strictly local data areas are located in a memory which is local to each processor. In such a way, local memory access will be faster than shared memory access. The code area is read-only and strictly local so it can be placed in the local memory. Since the data stack is used to pass arguments to functions and to return the result, we will also consider it as a local data area. This will speed up the function calls, but note that the result placed on top of the stack will be needed by remote processors. To manage this situation, the result is copied to the shared memory when another processor requests it. A similar solution is taken when machine registers are consulted by other processors. Machine register are fast local memory areas which are typically accessed by the local processor, but which can also be accessed by remote processors. We already noticed that remote processors can take charge of the control of a local parallel computation by accessing local registers. Therefore, a mechanism to access remote registers is also considered. Of course, local access to the code, data stack, or register data areas is faster than access to the data areas placed in shared memory. The point that must be analyzed is the overall time spent considering remote accesses to local data areas.

Figure 7 depicts the basic shared memory system in which n processors with local memory are connected to the shared memory through several lines and buses. The paths, common to typical shared memory systems, are:

- the *data bus*, which drives data to and from the shared memory,
- the *address bus*, in which each processor deposits the address to be read or written,
- the *mode line*, indicating the read, write, or wait mode [11] ,
- the *request line*, which is activated by the corresponding processor that wants to access the memory, and
- the *acknowledge line*, which is activated by the memory system to allow access to the corresponding processor.

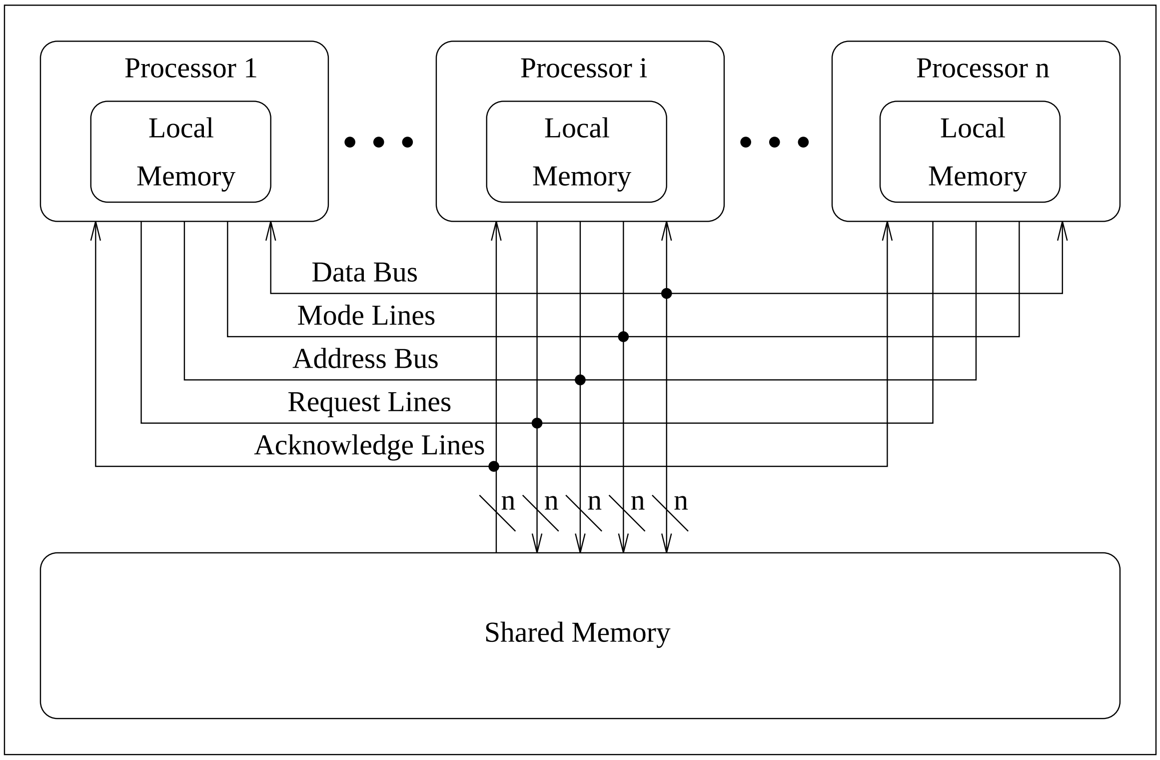

Figure 7: Shared Memory System

We have designed a First-Come-First-Served protocol that grants the access control to the first processor requesting the usage of the data bus. If several processors request

[11] Read and write modes are used to denote respectively a read or write request, while the wait mode is used to denote a wait on a critical section.

access at the same time, then this protocol behaves in a round-robin way. With this protocol we ensure that all the processors have the same priority in accessing the data bus.

Mutual exclusion management for critical sections is handled explicitly in our model. The usual semantics of semaphores has been specified, providing a wait statement on critical sections (i.e., a parcall frames) and a signal statement. We have extended the semantics in order to cope with kill interrupts, so that whenever a kill notification is received by a given processor, the execution of this processor is resumed in order to attend the kill interrupt.

6 VHDL Specification

In this section we present the VHDL specification of the parallel system. Firstly, we will briefly introduce the VHDL specification language. Then, we will present the shared memory system showing and its components, how they are connected, and how they are specified.

6.1 The VHDL Language

VHDL (VHSIC Hardware Description Language, where VHSIC is an abbreviation of Very High Speed Integrated Circuits) is a hardware description language, developed with the support of the US Department of Defence in order to gain uniformity in the description of design specifications [12, 3]. It has now become IEEE standard 1076-1987, revised in 1992.

VHDL was designed to fulfill a number of needs in the design process. Firstly, it allows the behavioural specification of the design using familiar programming language platforms. Secondly, it allows the structural description of a design, i.e. the decomposition into sub-designs and their interconnection. Thirdly, as a result, it allows a design to be simulated before being implemented, so that designers can quickly compare alternatives and test for correctness without the delay and expense of hardware prototyping.

A VHDL program is a set of concurrent processes communicating by means of signals, following an event-driven model. The behaviour of processes is described by means of a sequential algorithm making use of common imperative constructions. To handle process communication, the language provides the `WAIT` and the *signal assignment* (`<=`) statements. The signal assignment statement projects future values for signals. The general syntax is:

```
signal <= expression AFTER time expression;
```

The `expression` is evaluated and the result is scheduled to become the current value of the `signal` after the delay indicated by `time expression`. The `WAIT` statement allows to suspend the execution of a process, and to express the conditions for its resumption. The general syntax for a wait statement is:

```
WAIT ON sensitivity list UNTIL condition FOR time expression;
```

The logical `condition` is evaluated whenever an event happens in a signal which is in the `sensitivity list`. If the result is false, the process remains suspended, otherwise it is resumed after the time-out interval denoted by `time expression`. The time is only increased when a `WAIT` statement is executed.

The general syntax of a process is:

```
name : PROCESS declarative region
         BEGIN sequential statements
         END PROCESS;
```

It has a declarative region delimited by the reserved words `IS` and `BEGIN` where a set of data types, subprograms, and variables can be declared. A VHDL program is a set of processes concurrently executed following an event-driven model. Each process in a VHDL program represents a component or subsystem of the simulated hardware system. The simulation starts with an initialization phase, and then proceeds by repeating a simulation cycle. In the initialization phase, all signals are given initial values, the simulation time is set to zero, and the sequential algorithm of each process is executed until a wait statement is found. During the execution of a process, future values can be projected for some of the output signals of the corresponding part of the design by means of the signal assignment statement. In a simulation cycle the time is advanced to the next value when a signal changes or a process resumes. Each resumed process is executed from the next statement after the wait that caused its suspension until the next wait statement is found. If there are no more scheduled changes, the whole simulation is completed. Figure 8 shows a system consisting of a central processing unit (`CPU`) and a memory. The two sub-systems are connected by two data buses (`DataIn`, `DataOut`), an address bus (`Address`) and a control bus (`Read`, `Write`, `Ready`). The corresponding VHDL description is shown below.

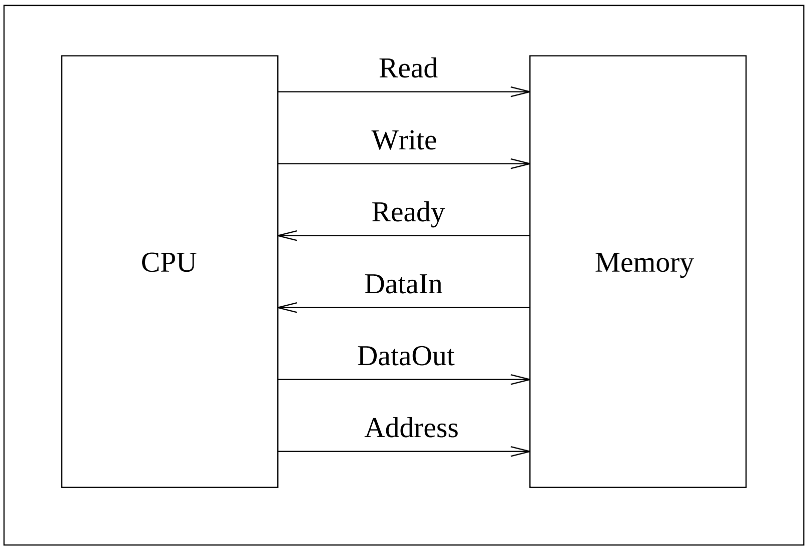

Figure 8: Processor system

```
Memory :  PROCESS
            BEGIN
               ...
               DataIn  <= ... ;
               ...
               Ready   <= ... ;
               ...
               WAIT ON Read, Write, DataOut, Address;
            END PROCESS Memory;
```

```
CPU :     PROCESS
             BEGIN
                ...
                Read    <= ... ;
                ...
                Write   <= ... ;
                ...
                DataOut <= ... ;
                ...
                Address <= ... ;
                ...
                WAIT ON DataIn, Ready;
             END PROCESS CPU;
```

The program consists of two processes, one for each component of the system, connected by signals. Each process is sensitive to the input signals and schedule transactions on its output signals. In VHDL it is possible to represent hardware components in which several outputs are connected forming buses. To achieve this, the language has special signals named resolved signals. These signals can be assigned from several processes at a time. A resolution function to resolve the way the signal changes must be supplied. Each time a process gives a new value to a resolved signal, the resolution function is called with the values that each process gives to that signal. The result of the resolution function is the effective value of the signal in that simulation time.

It is also possible to use a previously described digital circuit as a part of another by means of the component statement.

6.2 The Shared Memory System

We are interested in a low-level specification that allows to measure timings regarding the behaviour of the parallel system. This is why we have specified the system in such a way that, thanks to the VHDL's temporal model, we were able to specify the access memory times and evaluate the system in a refinement level so that the time granularity is in terms of register access time level. This refinement level allows us to evaluate the system at a really low level, where we can try out several design decisions such as different memory models (shared, interleaved, distributed, ...), cached systems, scheduling strategies, and so on. The aim is to get results as close to the actual system as possible in order to have close-to-real measurements before implementing the actual hardware system. Following this guideline, we have defined the memory access times for each data area (local access, shared memory access, and local data area remotely accessed) and have imposed the delays in the memory system by means of the VHDL statement `WAIT FOR memory access time;`. The simulation of such a specification provides the computation time of the parallel system, that can be compared with the computation time of the sequential system, which has been also specified in a similar way.

As stated in Section 5 we are interested in a system with a memory shared by a set of processors, forming a network. We map a machine to a processor, and the memory to the shared memory. We have designed the shared memory system as an asynchronous system in which the memory and each processor component are synchronized by means of the `Request` and `Acknowledge` lines. We consider three main categories in the shared memory system: the processor component, the memory component, and the paths linking them (i.e., the data and address bus, and the control lines). The processor and the memory are declared as VHDL components which allows easily us to declare as many processors connected to the memory as a constant indicates. The system can therefore be tested easily for different numbers of processors. Such processors perform communication through the data paths, that are naturally specified as VHDL signals.

The data paths are specified with the following VHDL signal declarations:

```
SIGNAL AddressBus: t_AddressBus;
SIGNAL MemProcDataBus: t_DataBus;
SIGNAL ProcMemDataBus: t_DataBus;
SIGNAL ModeLines: t_Modes;
SIGNAL RequestLines: t_Requests := (OTHERS => FALSE);
SIGNAL AcknowledgeLines: t_Acknowledges := (OTHERS => FALSE);
```

`AddressBus` stands for the address bus. For simplifying purposes, the data bus depicted in Figure 7 has been split into two data paths, namely `MemProcDataBus` and `ProcMemDataBus`, which stand for the outgoing data from the memory and the outgoing data from the processor, respectively. `ModeLines` represents the set of mode lines connecting each processor with the memory. `RequestLines` represents the set of request lines which are enabled by each processor requesting access to the memory. `AcknowledgeLines` represents the set of acknowledge lines connecting the memory with each processor. Only one acknowledge line is enabled at a time by the memory, therefore allowing access to the corresponding processor.

The identifiers above starting with `t_` stand for the type associated with the corresponding signal (typically, high level enumerated data types regarding possible values). The expression `:= (OTHERS => FALSE)` stands for the initialization phase of the corresponding signal. In our case, the intended meaning is the resetting of the requesting lines, as well as those acknowledging ones.

The interface parts of both processor and memory are considered next. Below, the VHDL ports that represent the sockets to which buses and lines (specified as signals) are connected, are shown.

```
COMPONENT processor
  GENERIC(Id : t_ProcId);
  PORT(InData: IN t_Data;
       OutData: OUT t_Data;
       Address: OUT t_Address;
       Mode: OUT t_Mode;
       Request: OUT t_Request;
       Acknowledge: IN t_Acknowledge);
END COMPONENT;
```

The `GENERIC` port is intended to provide an identifier to each processor, and the `Reset` socket is intended to initialize the processor at start-up.

In a similar way, the memory component is defined in VHDL as:

```
COMPONENT memory
```

```
  PORT(InDatas: IN t_DataBus;
       OutDatas: OUT t_DataBus;
       Addresses: IN t_AddressBus;
       Modes: IN t_Modes;
       Requests: IN t_Requests;
       Acknowledges: OUT t_Acknowledges);
END COMPONENT;
```

In the specification of the memory component, each socket is assumed to be connected to as many lines or buses as there are processors in the network. This approach simplifies the connections of data paths.

The specification of the components connection by means of the data paths is declared in the structural VHDL architecture shown below which links the sockets with the corresponding data paths.

```
BEGIN

  shmemo: memory   PORT MAP(InDatas      => ProcMemDataBus,
                            OutDatas     => MemProcDataBus,
                            Addresses    => AddressBus,
                            Modes        => ModeLines,
                            Requests     => RequestLines,
                            Acknowledges => AcknowledgeLines);

  processors: FOR i IN 1 TO c_numberofprocessors GENERATE

    proc_i: processor
                   GENERIC MAP(Id          => c_Ids(i))
                   PORT    MAP(InData      => MemProcDataBus(i),
                               OutData     => ProcMemDataBus(i),
                               Address     => AddressBus(i),
                               Mode        => ModeLines(i),
                               Request     => RequestLines(i),
                               Acknowledge => AcknowledgeLines(i);
  END GENERATE processors;

END structural;
```

The memory component embodies not only the data management (by serving read or write requests) but also the access protocol. Although Figure 7 depicts the local memory embodied in each processor (as well as the machine registers), in the specification we have embodied it in the memory. The reason is the following: since the data stack and register local data areas are to be remotely accessed, we have abstracted this mechanism in such a way that the memory system will be responsible for the management of those siblings, keeping the processor system free of it. Therefore, the memory system will serve all the data requests, but retains the conceptual behaviour of the local data areas, so that the access to them will not interfere with the access to the global data areas. We show below the body of the memory process that implements the intended behaviour.

```
    BEGIN
    WAIT ON requests;
    attend_local_requests;
    l_memory:
    LOOP
      IF no_pending_requests THEN
        EXIT l_memory;
      END IF;
      select_asker(v_PriorityArray, v_Target);
      attend_request(v_Target);
    END LOOP l_memory;
  END PROCESS;
```

The first statement implies a wait on the change of the `requests` signal. Whenever any processor sets its own request line, the change is noticed by the memory. After such a change is noticed, the memory first attends the local requests (`attend_local_requests`), which do not need to be refereed by the access protocol, because they are *per se* local requests. Afterwards, the memory system scans the active requests deciding which request to serve (`select_asker`), finally serving it (`attend_request`). With the use of the `LOOP` statement, all the eligible requests at a given time are attended, and the information needed by the access protocol is held in `v_PriorityArray`. The process delimiter `BEGIN` and `END PROCESS` define an overall loop, i.e., when all the request have been served, the process continues with the first statements and waits for further requests.

The extended behaviour of semaphores is specified in the `select_asker` procedure. Whenever the memory component detects a kill notification for the processor requesting a wait statement, the memory component serves the request returning a kill notification to the processor, respecting the protocol policy.

Below, we present the VHDL specification corresponding to the read operation performed by a processor.

```
Mode <= readMode;
Address <= ... ;
Request <= TRUE;
WAIT UNTIL Acknowledge = TRUE;
v_Data := InData;
Request <= FALSE;
WAIT UNTIL Acknowledge = FALSE;
```

When a read operation is requested by any processor, the `Mode` and `Address` lines are set with the corresponding values. Then, the `Request` line is set. The processor will wait until the memory decides to serve such a request. The (shortened) synchronization part of the memory is as follows.

```
outdatas(v_Target) <= ... ;
acknowledges(v_Target) <= TRUE;
WAIT UNTIL requests(v_Target) = FALSE;
WAIT FOR v_Delay;
acknowledges(v_Target) <= FALSE;
```

Once the acknowledge has been noticed by the processor, it can pick up the requested data from the socket `InData`, and then it resets the `Request` line. The memory, in turn, is synchronized again by the resetting of such line, and then, the corresponding delay is taken into account through the `WAIT FOR v_Delay;` sentence. Finally, the processor can resume its computation when the `Acknowledge` line is reset by the memory. From this point, the memory can attend to other requests.

This scheme forms the basics for controlling the access to shared data areas. It has been upgraded in order to serve the local memory accesses as well. The key to such an upgrade is to prevent the time constraint which occurs at the memory being transferred to the processor component so that it only affects the processor. It is specified by:

- avoiding time delays for local data areas in the memory, and
- imposing time delays for local data areas in the processor after any read or write operation.

The narrowing unit is specified in the processor specification. The body of the process specification is outlined below.

```
BEGIN
  printTraceStart;
  initProcessor;
  l_control_loop:
  LOOP
    read(RM, v_Data);
    IF v_Data.value = cod(stop) THEN
      EXIT l_control_loop;
    END IF;
    printDebugInfo;
    fetch(v_OpCode);
    CASE v_OpCode IS
      WHEN InitBam =>
        .
        .
        .
    END CASE;
  END LOOP l_control_loop;
  printTraceStop;
  WAIT;
END PROCESS;
```

After an initialization phase, the label `l_control_loop` denotes the beginning of the main control loop, which consists of:

- the exit condition based on the running mode register *rm*,
- fetching the current instruction, and
- a case selection for identifying and starting the execution of the current instruction. All the machine instructions are implemented at the case branches.

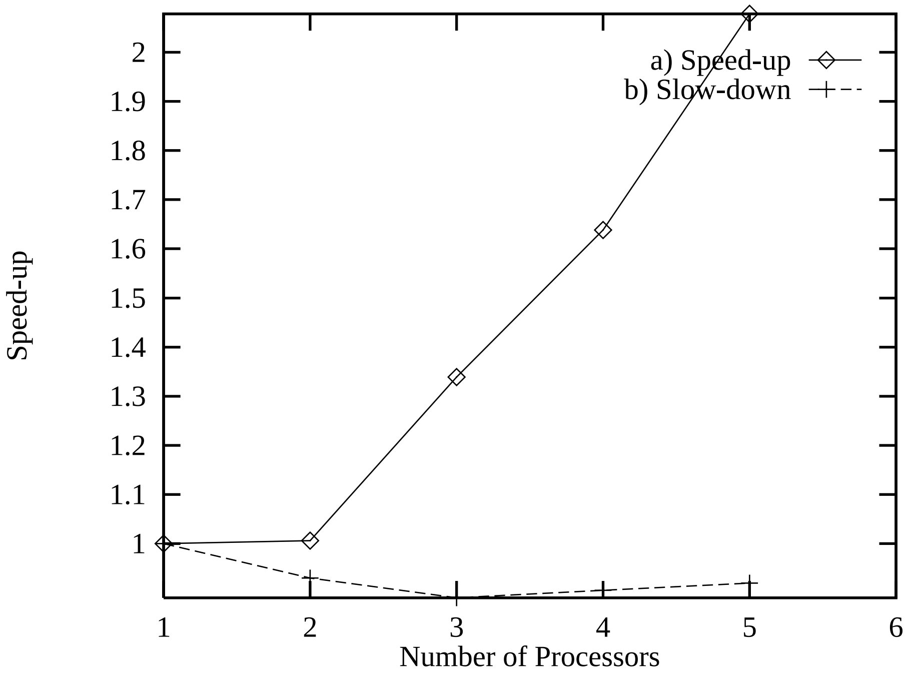

Figure 9: Speed-up vs. Number of Processors

After the main loop, the `WAIT` instruction ensures that the VHDL process stops.

The current version of the specification deals with debugging as well as post mortem analysis which has been provided through the procedures `printDebugInfo` for debugging and `printTraceStart`, `printTraceStop`, `printTraceFork`, `printTraceJoin`, `printTraceStartTask`, and `printTraceFinishTask` which supply the needed information for post mortem analysis using the visualizing tool VisAndOr [5].

7 Preliminary Results

In this section we give some preliminary results of the current VHDL implementation.

Figure 9 depicts the performance of the system in terms of speed-up obtained running the fibonacci program on different number of processors. We have noted that programs with not enough granularity show no speed-up, but slow-down, as one would expect (see Figure 9-b). This is clear due to the extra cost induced by the parallelism management.

Further improvements of the profiling procedure will deal with more refined statistics. For instance, we may consider the dead times for accessing the memory, the processors that remain idle due to the precedence condition, the time spent in the parallel related operations, and other factors which will help us to tune the system for further efficiency.

8 Conclusions and Future Work

We have presented a computational model for the parallel execution of Babel that relies on a more efficient memory management than in previous approaches. Another important extension is the optimized handling of the parallel execution of non-strict functions, a very essential concept of functional programming. All these ideas have been incorporated in an abstract model, assuming a shared-memory multiprocessor system. Further refinements have delivered a low level specification in VHDL and a prototype implementation. Currently we are studying its behaviour and refining the time measurement to obtain results closer to an actual parallel system. We have tested the system with a small set of benchmarks that shows the applicability of our procedure in the design of the parallel system.

Further investigation will deal with several topics. For instance, the study of interleaved, distributed, and cached memory models. An interesting topic is the integration of the distributed and the shared memory model, together with the combined exploitation of And-parallelism and Or-parallelism, since And-parallelism is well suited to exploitation with shared memory clusters, while Or-parallelism can be better exploited on a distributed network. Another topic is the extension of the non-strict parallel model to deal with lazy narrowing.

References

[1] M. Alpuente and M.J. Ramírez. The Logic + Equational Europa Environment and its Application to the Rapid Prototyping of Database Applications. In *Workshop on the Integration of Functional and Logic Programming*, Granada, Spain, September 1990.

[2] M. Bellia and G. Levi. The relation between Logic and Functional Languages: A Survey. *The Journal of Logic Programming 3:*, pages 217–236, 1986.

[3] J. M. Bergé, A. Fonkoua, S. Maginot, and J. Rouillard. *VHDL Designer's Reference.* Kluwer Academic Publisher, 1992.

[4] P.G. Bosco, E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. A Complete Characterization of K-LEAF, a Logic Language with Partial Functions. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 318–327. IEEE Comp. Society Press, 1987.

[5] M. Carro, L. Gómez, and M. Hermenegildo. Some Paradigms for Visualizing Parallel Execution of Logic Programs. In *1993 International Conference on Logic Programming (ICLP)*, 1993.

[6] C. Codognet and Philippe Codognet. Non-deterministic Stream AND-Parallelism based on intelligent backtracking. *Proceedings of the International Conference of Fifth Generation Computer Systems, ICOT*, 1984.

[7] J. Conery. *Parallel Execution of Logic Programs.* Kluwer Academic Publisher, 1987.

[8] D. DeGroot. Restricted And-parallelism. *Proceedings of the International Conference of Fifth Generation Computer Systems, ICOT*, 1984.

[9] D. DeGroot and G. Lindstrom. *Logic Programming: Functions, Relations and Equations.* Prentice Hall, 1986.

[10] M.V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

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

[12] Institute of Electrical and Electronic Engineers, Inc. *IEEE Standard VHDL Language Reference Manual*, March 1988.

[13] H. Kuchen and W. Hans. An AND-Parallel Implementation of the Functional Logic Language BABEL. In *Workshop on the Integration of Functional and Logic Programming*, Granada, Spain, September 1990.

[14] H. Kuchen, R. Loogen, J.J. Moreno-Navarro, and M. Rodríguez-Artalejo. Graph Narrowing to Implement a Functional Logic Language. Technical Report DIA 92/4, Departamento de Informática y Automática, UCM, 28040 Madrid, Spain, 1991.

[15] H. Kuchen, J.J. Moreno-Navarro, and M.V. Hermenegildo. Independent AND-Parallel Implementation of Narrowing. In *Book*, 1992.

[16] G. Lindstrom. Functional Programming and the Logical Variable. In *Proc. Symp. on Princ. of Programming Languages*, pages 266–280, New Orleans, 1985. ACM.

[17] R. Loogen. Stack-based Implementation of Narrowing. In *CCPSD, Tapsoft, LNCS 494*. Springer-Verlag, 1991.

[18] R. Loogen and St. Winkler. Dynamic Detection of Determinism in Functional Logic Languages. *Lecture Notes in Computer Science*, 1991.

[19] J.J. Moreno and M. Rodríguez-Artalejo. BABEL: a Functional and Logic Programming Language Based on a Constructor Discipline and Narrowing. In *Conference on Algebraic and Logic Programming, LNCS 343*, pages 223–232. Springer-Verlag, 1988.

[20] U.S. Reddy. Narrowing as the Operational Semantics of Functional Programs. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 138–151. IEEE Computer Society Press, July 1985.

[21] F. Sáenz and J. J. Ruz. Parallelism Identification in BABEL Functional Logic Programs. In *7th Conference on Logic Programming (GULP'92)*, Milan, Italy, June 1992.

[22] M. Sassin. Design of an Abstract Shared Memory Machine for AND-Parallel BABEL with Dependency Information. In *Workshop on the Integration of Functional and Logic Programming*, Granada, Spain, September 1990.

[23] D. H. D. Warren. An Abstract Prolog Instruction set. 309 Technical Note, SRI International, 1983.

[24] D. H. D. Warren. Or-Parallel Execution Models of Prolog. *Datsoft'87*, pages 243–257, 1987.