OM: A Virtual Processor for Parallel Logic Programs

John S. Conery
David M. Meyer*

CIS-TR-87-01
February 4, 1987

Abstract

The Warren Abstract Machine is the state of the art in implementation technology for Prolog. Its success is based on an instruction set that supports unification, clause selection, and the management of variable binding environments. Prolog is a sequential programming language, and many of the operations of the WAM rely on the standard, depth-first execution order of Prolog programs. This paper introduces OM, a virtual processor for parallel logic programs. The processor is being designed so that a collection of OM processors will be suitable for a large scale, non-shared memory multiprocessor. Some of the features of the WAM, such as instructions for unification, are present in OM, but other aspects, such as control instructions and the management of binding environments, have been redesigned to support programs of an abstract parallel execution model.

Submitted to ASPLOS-II

Department of Computer and Information Science
University of Oregon

*Supported by a Tektronix Fellowship

OM: A Virtual Processor for Parallel Logic Programs

1 Introduction

The idea of a virtual machine is an accepted and useful technique for implementing programming languages. There are many benefits, including portability to different host machines, ease of implementation, and flexibility for experimenting with different implementation techniques. Virtual machines have been designed for a number of languages, from the S-machines of the Burroughs B1700, to the Pascal P-machine, to virtual machines for more modern languages such as Smalltalk, Scheme, and Prolog. Another advantage, one that is especially true for languages such as Prolog that require complex interpreters, is that compilation to a virtual instruction set enables a much faster execution. A compiler can perform at compile time many of the operations that would be executed at runtime. The Warren Abstract Machine, or WAM, is an elegant design that enables compile-time optimizations of control decisions and memory allocation, leading to significant speed-up in the execution of Prolog programs [19].

Research in parallel execution of logic programming languages is now entering an interesting new phase, with many research groups studying implementation techniques. There are a number of competing abstract models for parallel execution of logic: committed choice AND-parallel models, such as Parlog [4], GHC [18], and Concurrent Prolog [17]; pure OR-parallel models [3]; and hybrid models that support both AND and OR parallelism, such as the AND/OR Process Model [7,5]. For each of these models, there are active research projects studying implementation techniques, and in most cases the work is centered on defining a virtual machine for programs of the abstract model.

This paper introduces a virtual processor architecture for programs of the AND/OR Process Model. The Opal Machine, or OM, is a virtual processor based on the implementation techniques used in Opal, a source-level interpreter for the AND/OR Process Model [12]. OM is a single processor, designed to work either by itself or in conjunction with other OM processors in a multiprocessor system. The OM incorporates aspects of the WAM that are not specific to sequential Prolog programs, along with new instructions that support variable bindings and control operations in a parallel system.

This paper is organized as follows. A high-level overview of logic pro-

gramming and the support provided by a WAM style machine will be presented in Section 2. In Section 3 the AND/OR Process Model is briefly described, and the requirements for runtime support are contrasted with the sequential Prolog machine. Some problems that arise when standard Prolog binding environments are adapted for parallel systems, and the solution used in Opal, are outlined in Section 4. OM itself and an example program are described in the final sections.

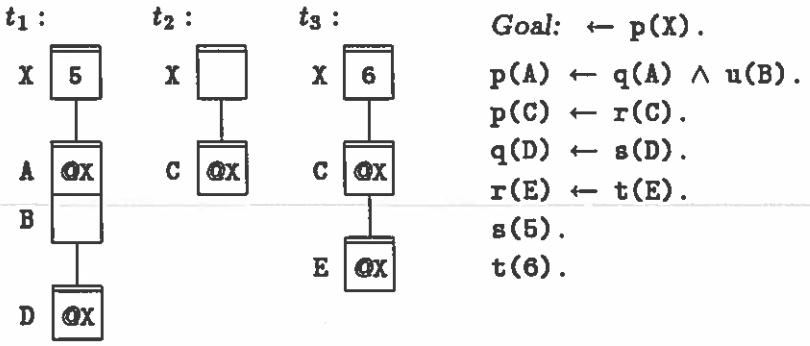
2 Machine Support for Logic Programs

The basic operation in a logic programming system is a logical inference. Typically, the inference step is based on *resolution*, an inference rule that has as an integral part a pattern matching operation known as *unification* [10,14]. Procedures of a logic program consist of a set of *clauses*. In programming language terms, each inference is a procedure call, where in order to invoke a procedure the call must match the pattern of the definition of the procedure. A call is sometimes called a *goal*, and executing the procedure is a means of solving the goal. What makes unification so powerful is the notion of variables in the patterns. The pattern matching operation is allowed to substitute a value for a variable, effectively binding the variable, in order to make the pattern match succeed. A key part of any implementation of a logic programming system is the technique for managing these variable bindings, since every inference requires access to bindings made in previous steps, and possibly creates new variables and bindings.

Another important aspect of implementing a logic programming system is the method for handling nondeterministic goals. In general, there may be more than one way to solve a goal, with a different solution corresponding to each different clause with a head that matches the goal. Systems that support nondeterminism, such as Prolog and Opal, must remember choice points when necessary, so that if the sequence of logical inferences along a chosen path lead to failure, an alternative can be tried. In Prolog, the alternatives are investigated sequentially; in an OR-parallel system, they are tried in parallel.

Figure 1 shows a simple set of clauses, an example goal, and the binding environments created in a typical Prolog system as the goal is solved. Space for variables is allocated from a stack, much the same as local variables of procedures are allocated in block structured languages. When a clause is selected as a potential match for a goal, the system allocates a *binding*

When a clause is called, a binding environment for the variables of the clause is pushed on the stack (which grows down the page). Environments are not popped until the clause fails. The symbol @X means the slot is bound to a pointer to slot X, meaning the two variables are to be considered the same variable. The first snapshot, on the left, shows the stack just after the call to `s(D)` *and before* `u(B)` *is attempted. At* t_2, `u(B)` *has failed and* `q(A)` *failed on retry, and* `r(C)` *is about to be called; note* `X` *is an unbound variable again. The right stack is the final stack, after the goal succeeds.*



Not shown are the two other stacks that give this binding representation technique its name, the "three-stack" model. A trail stack is used to hold backtracking information (this is where the system found the information to reset `X` *to unbound at* t_2*), and a heap is used to store instances of complex structures.*

Figure 1: Environment Stack for Sequential Prolog

environment, containing room for each variable of the clause, and unification binds the slots of the environment as necessary to complete the match. In general, slots from the environment of the call can be modified, as well as slots from the environment of the called procedure. If a unification fails, the system backs up to the last choice point, popping any stack frames created since the choice point.

The figure shows the system at three different points in the computation. In the first snap shot, the goal for p is being solved by the first clause for p, which has lead to a call to q, and from q to s. This example illustrates an interesting feature of logic programming. When two unbound variables are unified, the system does not assign either one a value, as such, but represents

the fact that the two variables have been "unified together." Operationally, this means a binding for one variable automatically becomes a binding for the other variable. This is implemented by binding one variable to a pointer to the other. In the figure, we see that A was unified with X, and then when D was unified with A it also became synonymous with X. The call to s unifies D with the constant 5, so the slot for X is bound to 5, binding all three variables simultaneously.

The second snap shot, at time t_2, shows the environments after the system failed to solve u in the first clause for p (when s was successfully solved, it meant q was solved, but the solution of p depends on both q and u), and has backed up and started to use the second clause for p. The third snap shot shows the final state, after the system has succeeded.

The WAM supports this style of computation in three different ways. First, and perhaps most important, there are instructions that perform the individual binding and accessing steps of unification. A Prolog compiler can do the top level decisions of unification at compile time, and generate special instructions to "finish the job" at runtime. The time savings come from the fact that the binding and accessing instructions are tailored specifically to each clause; there is little overhead in testing general cases. Second, the instructions are defined to work with respect to the style of binding environment illustrated in Figure 1. Addressing modes, dereferencing of pointers, and other concepts are defined with respect to the "three stack" model of binding environments. Third, control decisions are optimized. The first step in finding a clause to match with a goal is implemented by making a call to the address that stores the first clause with a head that can possibly match; in an interpreted system this requires a search or possibly a hash table operation. Control in nondeterministic programs is enhanced, as well, by having instructions that build choice points on the stack, such that if and when a unification fails, the system can quickly find the beginning of the next alternative in the sequence of logical inferences.

3 The AND/OR Process Model

The overall view of the execution of a Prolog program is of a sequential, depth-first search of a tree of derived goals. There are many ways to speed up the execution through parallel operations. The AND/OR Process Model provides an abstract framework for exploiting many of these sources of parallelism.

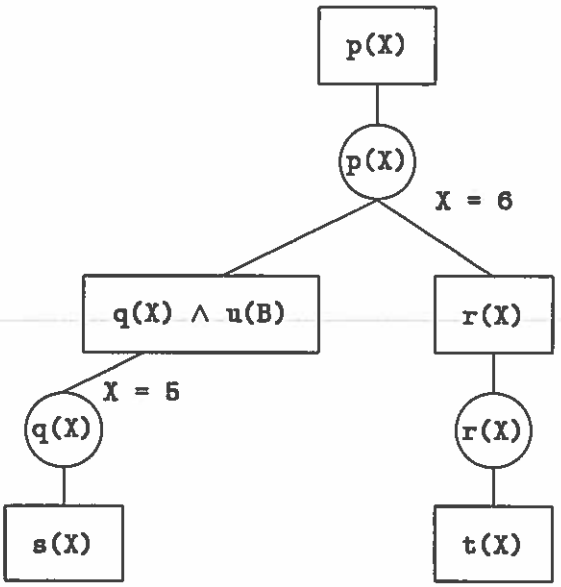
In the AND/OR Process Model, logic programs are interpreted by a set of independent, asynchronous objects that communicate via messages [7,5]. The status of an object is represented by internal state variables that are updated only when the object receives a message from another object. State transitions are atomic operations. When implemented at the level of OM instructions, a state transition will be carried out as an uninterrupted sequence of virtual instructions.

The two types of objects in the AND/OR Process Model are known as *AND processes* and *OR processes*. A process is created with an initial state that represents a portion of a logic program that needs to be solved. In general, a process creates other processes to solve subgoals, assemble results from the subgoals, and pass the results back to its parent. A process terminates after it has generated all possible solutions to its goal. An AND process is created to solve a conjunction of goals, such as the user's initial goal statement. An AND process solves each of the individual goals in the conjunction by creating a separate OR process for each; the OR processes coordinate solutions based on alternative clauses for their goals, and report results back to their parent AND process one at a time.

Parallel execution in this model comes from two different sources. AND parallelism is exploited if the goals in a conjunction are independent and can be solved in parallel; in this case an AND process creates multiple OR processes in one step and they execute in parallel. OR parallelism is exploited if there is more than one clause that can be used to solve a goal. If these clauses are not simple assertions, an OR process will create AND processes to work on the bodies of the clauses in parallel. The logic program of Figure 1 is reproduced in Figure 2 along with a description of the parallel processes that are created to solve the initial goal.

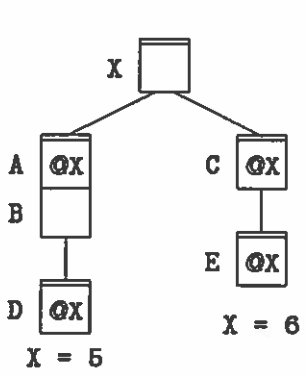
The AND/OR Process Model has the same theoretical roots as Prolog, namely resolution proof of statements of clausal logic, and an implementation of the model has to support efficient unifications. Unification is implemented in the OM the same way it is in the WAM: top level decisions are made at compile time, and runtime accessing and binding of variables is done through individual instructions. The first major difference between WAM and OM stems from the differences in the structure of the binding environments. Problems that arise when the standard Prolog binding representations are employed in parallel systems, and our approach to solving these problems, are discussed in the next section. The other major difference between WAM and OM is in control flow. The instructions used to create and call asynchronous processes will be described in Section 5.

Boxes represent AND process, circles represent OR processes. An AND process is solved when all its descendant OR processes succeed, but an OR process succeeds if any of its descendants succeed.



In this snapshot, the OR process for `p(X)` *is about to receive a success message from the process for the second clause for* `p`*. The AND process for the first clause is about to receive a success message from its first subgoal and start an OR process for its second subgoal, which will fail. If there was a way to solve* `u(B)`*, the AND process would succeed and send a success back to the OR process for* `p`*. In the AND/OR Process Model, the OR process buffers this solution until its parent sends a redo message, signifying it wants another solution. The binding environments discussed in Section 4 developed for the AND/OR Process Model manage the conflicting bindings for* `X`*.*

Figure 2: Parallel Processes



Goal: ← p(X).

p(A) ← q(A) ∧ u(B).
p(C) ← r(C).
q(D) ← s(D).
r(E) ← t(E).
s(5).
t(6).

In an OR-parallel system, a new process is started at a choice point in the program, such as the call to p(X) *here. If new processes share the existing stack, each needs its own copy of variables which are unbound at the time of the fork. In this example the processes share the frame for the original goal, and eventually generate conflicting bindings for* X.

Figure 3: Environment Stack in OR-Parallel System

4 Binding Environments for Parallel Logic Programs

One way to implement binding environments in an OR-parallel system is to allow new processes to share information from their common ancestors. If a process reaches a choice point late in the computation, it can spawn new processes, one for each alternative in the choice point, and a significant savings in space can be realized if each new process shares the bindings made before the choice point. However, there is a potential problem, as illustrated in Figure 3. The figure shows our example program and what happens if parallel processes share the binding environments in existence at the time the processes split. In this example, the environment of the goal, with the variable X, is the only existing environment when two processes for p are started.

The problem is caused by the fact that ancestor slots may be unbound at the time new parallel processes are spawned, and each new process may attempt to bind the unbound variable to conflicting values. There are a number of techniques for avoiding the conflict. Ciepielewski and Haridi were

the first to tackle the problem [3], describing a system that let processes share fully bound frames but copy frames with unbound slots. This works because logic languages are single assignment languages, and a descendant can never change a binding made by one of its ancestors. Related techniques have been developed by Borgwardt [2], Lindstrom [11], D. S. Warren [20], and others. All rely on the construction of auxiliary data structures, enabling a process to keep its own copy of a shared unbound ancestor variable.

When the three stack model is adapted in this way, so that parallel processes share information, the underlying machine must have a single address space, either in a shared memory multiprocessor, or maintain a single address space in physically disjoint memory modules. When a variable is bound to a pointer to a cell farther back on the stack, the address of the ancestor may be nonlocal, so a "boudoir" configuration of processors and memories my be penalized fairly often. When many parallel processes check the binding status of their common ancestor, the address of the ancestor may become a "hot spot" of the type investigated by Pfister and Norton [13], so a "dance hall" configuration of processors and memories may also have problems.[1] Empirical evidence for such a pattern of memory references can be seen in plots produced by Ross and Ramamohanarao [15]. They plotted the addresses of memory references *vs.* time in a Prolog interpreter, in order to measure locality of reference. They found clusters of locality near the top of the stack and at the base of the stack, where the unbound shared ancestor variables would be found.

For the implementation of Opal, we developed a new scheme for maintaining variable bindings. We had two goals: develop a scheme that does not rely on a shared memory, so that binding environments can be located and moved anywhere in a non-shared memory system; and develop a scheme that would be suitable for the requirements of the AND/OR Process Model. The resulting technique is called *closed environments* [6]. A closed environment is one that has no pointers that lead directly or indirectly to the environment of any other clause.

After every unification in Opal, an environment closing operation is applied to the environment of the called procedure, transforming it into closed form. The newly closed environment is then used in the next round of unifications. Since a closed environment cannot contain pointers to shared ancestor variables, conflicts between parallel processes are avoided. After the subgoal is solved, the environments are closed in the opposite direc-

[1]Most of the references will be reads, however, so a network that combines references may handle this problem.

tion, with the parent closed with respect to the descendant. This serves two purposes: the parent environment is now in closed form, ready to be used when solving siblings of the recently solved goal, and it incorporates bindings from the solution of the descendant into the parent's environment, avoiding the need for a "back unification" step [21]. Virtual instructions to carry out the steps of the closing operation have been defined as part of the OM instruction set.

5 The OM Virtual Processor

5.1 Overview

OM is a virtual *processor*. It has been designed to work either as a single processor in a von Neumann architecture, or as one of many processors in a shared memory machine such as the NYU Ultracomputer [9], or as the processor in a processor-memory pair of a hypercube or similar non-shared memory machine [16]. Since the closed environment technique does not rely on a common memory space, and there will be no pointers to ancestor variables far back in the stack, OM should potentially work well in a large-scale, non-shared memory multiprocessor.

An OM processor maintains a queue of messages and the states of a set of AND and OR processes. It continually executes an algorithm where it removes a message from the queue, activates the corresponding process, and performs the indicated state transition. Each state transition corresponds to one atomic step of an AND or OR process, and potentially generates new messages and processes. In a system with more than one OM processor, processes and messages will be distributed so that state transitions can be executed in parallel. The current implementation is for a single processor-memory pair. At this time, we are concerned mainly with the correctness of the instruction set, and the efficiency with which it performs unifications, builds and accesses closed environments, and carries out the control operations of a parallel logic program.

5.2 OM Registers

Every unification works with two environments. The environment of the call is known as a *top* environment, and the environment of the called clause is known as a *bottom* environment. The names are based on a graphical method (known as Ferguson diagrams) for representing unifications in a

Prolog interpreter, where environments are represented as top or bottom half-circles [8]. A third, temporary, environment, known as X, is sometimes used by an OR process.

The slots of an environment are referred to by the environment name and an index; for example, `T0` is the first slot in the current top environment, `B1` is the second slot in the current bottom environment, and so on. We will sometimes refer to the slots of an environment as "the T registers" or "the B registers." There are many ways to implement these registers. They could be a fixed set of internal processor registers, or special high-speed memory, or simply locations in main memory pointed to by registers named TE and BE. There are many factors that will go into deciding which technique to use. The granularity of process steps is small to medium, and there will be overhead in loading environment registers when a process is activated, so the total overhead from task switches must be taken into account (this is discussed in more detail in the summary). On the other hand, many unifications fail. If the bottom environment is in registers, there is no need to allocate room for it in memory, then abandon it when the unification fails; an environment will be put in main memory only after unification succeeds and if the environment must survive across process activations. A third component of the decision is related to the environment closing operation. During some steps, an environment must be extended with one or more new slots. If the environment is in registers or a fixed block of dedicated, fast memory, it can easily be extended, and when it is saved the old location can be discarded and a new block allocated for the larger frame. Currently we are simulating the second solution, dedicated blocks of fast memory for each active environment.

As in the WAM, a set of argument registers, called *A registers*, will be used to hold the parameters passed in a procedure call. The code in an AND process that sets up a call to an OR process will use a series of put instruction to place the parameters in the registers. For each argument in the head of a clause, there will be get instructions to perform the actual unification steps, binding variables when necessary. The OR process never modifies the A registers. If a unification step needs to bind a parent variable, the get instruction will follow a path through the A registers to a slot in the T registers. Since the A registers are not modified, the unification of each clause head can use the same A values. The A registers are saved in the state vector of the OR process, since later steps will need to use them when passing values back to the parent process.

Other registers used to define the state of the machine will be introduced

as necessary in the discussion.

5.3 Compilation Schema

A schema for compiled programs is shown in Figure 4. There are two classes of translations: those for AND processes and those for OR processes. Each procedure, defined to be a set of clauses that have heads with the same predicate symbol and same number of arguments, is compiled into an OR process. Each non-unit clause (a clause with one or more goals in its body) is compiled into an AND process. The compilation of AND and OR processes, and the new instructions to support each, will be discussed in the next two sections.

5.4 Compilation of an OR Process

The code for an OR process is invoked when an AND process creates a new OR process for a procedure. When the process is started, the A registers will contain the arguments passed by the parent AND process, and the parent's environment will be in the T registers. Each time a new OR process is started, it tries to unify the arguments in the A registers with the arguments of every potentially matching clause head in the procedure. For each matching unit clause (a clause with no subgoals in the body), the OR process creates a success message. For each matching non-unit clause, a new AND process is started. After all matching operations are done, the OR process is suspended until it receives another message.

The control instructions used in the compiled OR process are similar to the WAM `switch` and `try` instructions. `switch_on_term` examines the type of the first argument, and branches to a location within the body of the process where it will find the code for the clause head(s) whose own first argument(s) have that type. `Lv` is the address to go to if the passed argument is a variable; this means the call will match any compiled head, so all of them must be tried. `La`, `Ls`, and `Lx` are addresses to branch to if the passed parameter is an atom, structure, or list, respectively. In the schema for an OR process in the figure, there are three clause heads, starting at addresses `La`, `Ls`, and `Lx`.

When the input parameter is a variable, or there is more than one clause for a particular argument type, the OR process executes instructions that set up a sequence of unifications. In an eager OR process, all unifications are done as part of its first step, setting up as many parallel AND processes

Compilation of an OR process

```
proc:   switch_on_term          Lv,La,Ls,Lx
Lv:     next_alternative_else   L1
        make_top
La:     (get instructions)
        succeed
L1:     next_alternative_else   L2
        restore_top
Ls:     make_bottom             N1
        (get instructions)
        (close instructions)
        succeed
L2:     last_alternative
        restore_top
Lx:     make_bottom             N2
        (get instructions)
        allocate_top
        start_and               C1
        (close instructions)
        succeed
```

Compilation of an AND process

```
C1:     start_or
        succeed
C2:     (put instructions)
        start_or                G1
        (put instructions)
        start_or                G2
        succeed
```

Figure 4: Compilation Schema for OR and AND Processes

as possible. The next_alternative instruction specifies the address where where execution resumes after the current unification is finished, successful or not. The operand of this instruction is stored in the Continuation Register, or CR. next_alternative does not build a choice point data structure, the way the try instructions of the WAM do, since these structures are not used in the parallel machine.

When there are a number of alternatives to explore, each can potentially bind slots in the parent's environment in a different way, so each alternative needs its own copy of the parent environment. The make_top instruction copies the T registers to the X registers. At the start of each successive unification, a restore_top instruction copies the X registers to the T registers, making a new copy of the parent for that alternative. Note that if there is only one way to solve a goal, there is no need to copy the parent environment and set up a continuation: there would be no make_top and next_alternative instructions, and CR contains a value (set when the process starts) that indicates no more choices exist after the current unification is done.

The actual head of a clause begins with a make_bottom instruction, which initializes a bottom environment of the correct size in the B registers. The unification itself begins with the get instructions. If any of these fail, meaning an argument passed in from the parent does not unify with an argument of the called procedure, a failure occurs, and the machine either branches to the address stored in the CR register, or, if this is the last or only alternative, suspends the OR process (what happens when the process is suspended is discussed below).

If all of the get instructions are executed, the unification succeeds. If the clause is a unit clause, the OR process executes a a succeed instruction, without starting an AND process. The heads at addresses La and Ls in Figure 4 are examples of unit clause compilations. A succeed instruction creates a success message for the parent process. The current top environment is the argument of the success message. Execution resumes in the OR process at the address following succeed.

If there is a body for the clause, an AND process must be created to solve it (see the clause at address Lx in the example). The allocate_top instruction stores the T registers in the state vector of the OR process, so the bindings made to the top environment during unification will be available when the AND process returns a success message. The get instructions are implemented in such a way that the bottom environment is always in closed form after a unification, so all we have to do next is execute a start_and.

This instruction starts an AND process, using the current bottom environment as its initial binding environment. The address of the code for the AND process is the operand of the `start_and` instruction. Execution of `start_and` creates a new process descriptor, inserts it in the process queue, sends it a start message, and then continues execution in the OR process at the address currently in CR. The state vector of the OR process is updated to store the address following the `start_and` instruction; when the AND process returns a success message, the system restores the A and T registers, activates the OR process, and branches to this address.

When the head of a clause is *ground* (it contains only constants, no variables), nothing has to be done to the top environment before it is returned to the parent AND process in a success message. However, when there are unbound variables in the head of the clause, the corresponding `get` instructions may have bound a parent variable in such a way that the top environment has pointers that dereference to slots in the bottom environment. These slots have to be closed before the top environment is returned to the parent. A series of `close` instructions implement an "open-coded" environment closing operation, closing the top environment with respect to the bottom environment, so it can be sent back to the parent AND process in closed form. In the example, the head for `La` is ground, but the other two clauses had variables in the head.

The `close` instructions are the means for transmitting bindings from the descendant process back to the parent. If the parent passed an unbound variable as an argument, and it was unified with a variable `X` in the head of the clause, and `X` was bound by the solution of the descendant AND process, the corresponding `close` instruction will bind the parent variable to the value of `X`.

When all alternatives have been explored, the OR process is done with its first step. Before suspending itself, the OR process executes the protocol described in [7] to see if it should send a message to its parent. If no unifications succeeded, the OR sends its parent a fail message and terminates. If one or more success messages were generated by `succeed` instructions, one of the messages is sent to the parent, and the OR process suspends itself in gathering mode. If there are no success messages, but active AND processes, the OR process is suspended in waiting mode.

When a descendant AND process sends a fail message, the address of the continuation is removed from the state vector of the OR process. When all AND descendants have failed, and there are no more results waiting to be sent to the parent, the OR process sends its parent a fail message and

terminates.

5.5 Compilation of AND Processes

Currently we are not exploiting AND parallelism in the OM; control instructions have been defined for sequential AND processes only. These processes solve the bodies of their clauses in the same order as Prolog, working from left to right, and re-solving the most recently solved goal when a goal fails.

The compiled code of the body of a sequential clause consists simply of instructions to set up the arguments of the procedure call, and then start an OR process for the goal. The state vector of an AND process contains a fixed size array of records, one for each subgoal in the body of the clause, where each entry describes the progress begin made in solving that subgoal. `start_or` stores the process ID of the OR process solving a subgoal in the appropriate record, along with the address of the instruction to execute when the process sends a success message.

When an OR process sends back a success message, the environment stored in the message becomes the current environment for the AND process. Any put instructions that refer to variables of the AND process' environment will access slots of the newly returned environment.

If an OR process sends a fail message, the ID of process for the most recently solved subgoal is found in the state vector, and this process is sent a redo message. If that process sends another success, the environment passed in the success message becomes the new current environment for the AND process. It is interesting to note that each step of an AND process uses an environment returned by OR processes, and that no environment has to be stored in the state vector of the AND process.[2] In this implementation, an AND process is a control process, determining the order goals should be solved, while the actual execution steps are carried out in unifications in the OR processes.

The AND process fails when its leftmost subgoal fails, and it succeeds after the OR process for its rightmost subgoal sends a success message.

6 Example

Figures 5 and 6 show the compiled code for the program of Figure 1. Figure 5 contains the code compiled for the heads of the clauses used as

[2]This may not be true for parallel AND processes.

```
p/1:    make_top                1          % procedure for p
        next_alternative_else   L1
        make_bottom             2          % p(A) ← ...
        get_var                 B0,A0
        alloc_top               1
        start_and               L2
        close                   A0
        succeed
L1:     last_alternative
        restore_top             1
        make_bottom             1          % p(C) ← ...
        get_var                 B0,A0
        alloc_top               1
        start_and               L4
P1:     close                   A0
        succeed

q/1:    make_bottom             1          % procedure for q
        get_var                 B0,A0
        alloc_top               1
        start_and               L3
        close                   A0
        succeed

r/1:    make_bottom             1          % procedure for r
        get_var                 B0,A0
        alloc_top               1
        start_and               L5
        close                   A0
        succeed

s/1:    get_const               5,A0       % procedure for s
        succeed

t/1:    get_const               6,A0       % procedure for t
        succeed
```

Figure 5: OR process code for the program in Figure 1

```
goal/0: put_var          T0,A0    % ← p(X).
        start_or         p/1
goal1:  end

L2:     put_val          T0,A0    % ← q(A) ∧ u(B).
        start_or         q/1
        put_val          T1,A0
        start_or         u/1
        succeed

L3:     put_val          T0,A0    % ← s(D).
        start_or         s/1
        succeed

L4:     put_val          T0,A0    % ← r(C).
        start_or         r/1
        succeed

L5:     put_val          T0,A0    % ← t(E).
        start_or         t/1
        succeed
```

Figure 6: AND process code for the program in Figure 1

examples throughout the paper; this code is used to start an OR process. Figure 6 has the code compiled for the bodies of the clauses, used for AND processes. OM executes this program as follows: execution begins with the creation of an AND process for the top level query. This AND process, A_{goal}, begins at the label `goal/0`. First, the argument registers are loaded, and an OR process, O_p, is created to solve the literal `p(X)`. Since the code in Figure 6 is compiled for a Sequential-AND/Parallel-OR model, the current AND process (A_{goal}) is suspended and O_p is scheduled to run.

O_p begins running at the label `p/1`. First the top environment is saved for use by the other alternatives for `p/1`, which are linked together by `next_alternative_else` instructions. The first alternative represents the head of the clause `p(A)` ← `q(A)` ∧ `u(B)`. The top environment is copied

to the X registers, since there is another clause for p/1 that will be tried later, and a new bottom environment with slots for A and B is initialized. The get instruction results in a *link* (a relative pointer that dereferences to a variable [6]) pointing to the variable X in the top environment to B0, which is the slot for A. This link will participate in later closing operations. Notice that a copy of the top environment is not made until we are sure that unification will succeed. After unifying the arguments and saving the top environment, a new AND process, A_q, is created to execute the body q(A) $\wedge$ u(B). Since our current implementation is fully OR parallel, O_p retains control and continues execution at the label L1. O_p will eventually create an AND process A_r to solve $r/1$ and then suspend itself.

In this example A_q will eventually fail and A_r will succeed. If O_p receives a fail message from A_q while A_r is still active, O_p will suspend itself again after removing A_q from its state vector. When O_p receives a success message from A_r, O_p will be scheduled to run, and the environment in the success message from A_r will be installed as the bottom environment. O_p then resumes execution at the label P1, executing the appropriate close instructions. The result of these instructions removes the link from X to A and binds the slot for X in the top environment to the value of A in the bottom environment.

Having solved p(A) with the binding A = 6, O_p executes a succeed instruction. Since O_p was in waiting mode, it will send a success message containing the current top environment to its parent, and go into gathering mode.

Next, A_{goal} will be scheduled to run when it receives the success message. A_{goal} will resume execution at the label goal1. At this point, OM executes an end instruction, indicating that the computation has been successfully completed.

7 Summary and Future Work

The OM is a virtual processor designed to support parallel logic programs in a non-shared memory architecture. It is similar to the WAM in the way it performs open-coded unification, based on get and put instructions compiled explicitly for each different clause of the program. It differs from the WAM in two significant respects: it manipulates closed form binding environments, which have been designed to avoid the problems of the three-stack model in parallel systems, and the control instructions are based on

an asynchronous, object-oriented control structure.

The instruction set, as outlined here, requires a large degree of support from what are normally considered operating system functions. `start_and`, `succeed`, and other instructions that manipulate implied data structures such as the process queue have been presented as macroinstructions. They are currently implemented in "microcode," *i.e.* in the supporting interpreter. A much more desirable solution is to program these operations in a logic programming language as well, implementing them as sequences of OM instructions activated by traps. We are currently investigating methods for doing this, either by a special kernel mode language for system routines, or through techniques for incorporating history sensitive objects into logic programs.

The steps of the abstract execution model are medium-grain operations, and they are implemented faithfully in the OM. The step that does the most work is the step taken by an OR process when it first starts. A new OR process attempts unifications with the head of every clause in the procedure, starting AND processes for the bodies of each successful nonunit clause. There are a number of optimizations that are possible if this step, and other complex steps, of the abstract model are separated into smaller independent operations. One improvement is to allow an AND process to use a `call` instruction instead of `start_or` when a procedure is defined by a set of unit clauses. The overhead of an OR process that creates all answer frames in one step may not be justified in many cases. Another improvement is to cut down on the amount of OR parallelism. Many OR parallel programs describe combinatorially explosive search spaces, and to eagerly explore all branches in this kind of program in parallel would be foolish, since the number of processes could soon swamp the machine. The code for the head of a procedure should allow `start_and` to be implemented in a way that transfers control to the new AND process, instead of always coming back to the OR process where additional unifications are performed. Strategies for deciding when to do an eager OR-parallel search, methods for gradually switching to less eager search when the machine starts to become overloaded, and the effect of this implementation of `start_and` on the other aspects of OR processes, need to be explored further.

Carrying this idea further, and allowing the AND process to give control to the first OR process it creates, leads to a depth-first search. When there is an orderly transition between related processes, the overhead of a task switch is reduced, because one of the two environments will be in registers already. An idea we intend to pursue is to have a "track counter" that

counts down with each inference step; when it is 0, the machine will switch to another track, but when it is nonzero after a step the next process is the one that would be chosen in a depth-first search.

A "hardware solution" to the cost of context switches may also be possible. In the FAIM-1, a switching processor will run in parallel with an evaluation processor, setting up a parallel set of registers with values to be used by the next process [1]. Context switches occur in one machine cycle, by having the evaluation processor switch its attention to the other set of registers. This scheme would appear to work for OM, as well, since the environments that would be loaded by the switching processor are guaranteed to be independent of the environments that are being modified by the evaluation processor.

References

[1] Anderson, J.M., Coates, W.S., Davis, A.L., Hon, R.W., Robinson, I.N., Robison, S.V., and Stevens, K.S. The architecture of FAIM-1. *IEEE Computer 20*, 1 (Jan. 1987), 55–65.

[2] Borgwardt, P. Parallel Prolog using stack segments on shared memory multiprocessors. In *Proceedings of the 1984 International Symposium on Logic Programming*, (Atlantic City, NJ, Feb. 6–9), 1984, pp. 2–11.

[3] Ciepielewski, A. and Haridi, S. *Formal Models for Or-Parallel Execution of Logic Programs.* CSALAB Working Paper 821121, Royal Institute of Technology, Stockholm, Sweden, 1982.

[4] Clark, K.L. and Gregory, S. PARLOG: Parallel programming in logic. *ACM Trans. Prog. Lang. Syst. 8*, 1 (Jan. 1986), 1–49.

[5] Conery, J.S. *The AND/OR Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, Univ. of California, Irvine, 1983. (Computer and Information Science Tech. Rep. 204).

[6] Conery, J.S. *Closed Environments: Partitioned Memory Representation for Parallel Logic Programs.* Tech. Rep. 86-02, Univ. of Oregon, Feb. 1986.

[7] Conery, J.S. and Kibler, D.F. Parallel interpretation of logic programs. In *Proceedings of the Conference on Functional Programming Lan-*

guages and Computer Architecture, (Wentworth-by-the-Sea, NH, Oct. 18–22), ACM, 1981, pp. 163–170.

[8] van Emden, M.H. An interpreting algorithm for Prolog programs. In *Proceedings of the First International Logic Programming Conference*, (Faculté des Sciences de Luminy, Marseille, France, Sept.), 1982, pp. 56–64.

[9] Gottlieb, A., Grishman, R., Kruskal, C.P., McAuliffe, K.M., Rudolph, L., and Snir, M. The NYU Ultracomputer – Designing an MIMD shared memory parallel computer. *IEEE Trans. Comput. C-32*, 2 (Feb. 1983), 175–189.

[10] Kowalski, R.A. *Logic for Problem Solving.* Elsevier-North Holland, New York, NY, 1979.

[11] Lindstrom, G. OR parallelism on applicative architectures. In *Proceedings of the Second International Logic Programming Conference*, (Uppsala, Sweden, Jul. 2–6), 1984, pp. 159–170.

[12] More, N. *Implementing the AND/OR Process Model.* Master's thesis, Univ. of Oregon, 1986.

[13] Pfister, G.F. and Norton, V.A. "Hot spot" contention and combining in multistage interconnection networks. In *Proceedings of the 1985 International Conference on Parallel Processing*, (Aug.), IEEE, 1985, pp. 790–797.

[14] Robinson, J.A. A machine oriented logic based on the resolution principle. *J. ACM 12*, 1 (Jan. 1965), 23–41.

[15] Ross, M. and Ramamohanarao, K. Paging strategy for Prolog based dynamic virtual memory. In *Proceedings of the 1986 Symposium on Logic Programming*, (Salt Lake City, UT, Sep. 22–25), 1986, pp. 46–57.

[16] Seitz, C.L. The cosmic cube. *Commun. ACM 28*, 1 (Jan. 1985), 22–33.

[17] Shapiro, E.Y. *A Subset of Concurrent Prolog and its Interpreter.* Tech. Rep. TR-003, Institute for New Generation Computer Technology, Tokyo, Japan, Jan. 1983.

[18] Ueda, K. *Guarded Horn Clauses.* Tech. Rep. TR-103, Institute for New Generation Computer Technology, Tokyo, Japan, June 1985.

[19] Warren, D.H.D. *An Abstract Prolog Instruction Set.* Tech. Note 309, SRI International, Oct. 1983.

[20] Warren, D.S. Efficient Prolog memory management for flexible control strategies. In *Proceedings of the 1984 International Symposium on Logic Programming*, (Atlantic City, NJ, Feb. 6–9), 1984, pp. 198–202.

[21] Wise, M.J. A parallel Prolog: The construction of a data driven model. In *Conference Record of the Symposium on LISP and Functional Programming*, (Pittsburgh, PA, Aug. 15–18), ACM, 1982, pp. 55–66.