Relating Goal Scheduling, Precedence, and Memory Management in AND-parallel Execution of Logic Programs

M. V. Hermenegildo
Microelectronics and Computer Technology Corporation (MCC)
3500 West Balcones Center Dr.
Austin, TX 78759

Abstract: The interactions among three important issues involved in the implementation of logic programs in parallel (*goal scheduling*, *precedence*, and *memory management*) are discussed. A simplified, parallel memory management model and an efficient, load-balancing goal scheduling strategy are presented. It is shown how, for systems which support "don't know" non-determinism, special care has to be taken during goal scheduling if the space recovery characteristics of sequential systems are to be preserved. A solution based on selecting only "newer" goals for execution is described, and an algorithm is proposed for efficiently maintaining and determining precedence relationships and variable ages across parallel goals. It is argued that the proposed schemes and algorithms make it possible to extend the storage performance of sequential systems to parallel execution without the considerable overhead previously associated with it. The results are applicable to a wide class of parallel and coroutining systems, and they represent an efficient alternative to "all heap" or "spaghetti stack" allocation models.

Keywords: LOGIC PROGRAMMING, PARALLEL PROCESSING, AND-PARALLELISM, SCHEDULING, MEMORY MANAGEMENT, COROUTINING, PROLOG.

1 Introduction

The most promising strategy for increasing the execution speed of logic programs [14] presently appears to be the combination of advanced compiler technology with parallel execution. Several models have been proposed for the implementation of logic programs in parallel [7, 5, 16, 6, 15, 9, 2, 4, ...]. However, some issues which have already been the target of extensive optimizations in sequential systems and which very dramatically affect the

efficiency of a practical implementation are often devoted secondary attention in some models. One of these issues is *memory management*. In this paper, the relationship between *goal scheduling* and *memory management* in AND-Parallel "don't know" non-deterministic systems will be addressed. Clearly, the desirable characteristics to strive for are: minimization of idle processor time, storage optimization, garbage collection minimization, and load balancing, among others. Basic scheduling and memory management strategies will be presented, and it will be shown how the techniques used in sequential systems for storage economy and avoidance of garbage collection through the recovery of space during backtracking can be efficiently extended to parallel, multiple-stack systems.

Organization of the paper is as follows: section 2 reviews the relationship between goal precedence and storage management in *sequential* systems using a simplified memory management model. In section 3 this model is extended to support parallel execution and a distributed goal scheduling strategy is described. Section 4 then discusses the interactions between memory management and goal scheduling. Two basic problems which result from these interactions are discussed. A solution for these problems based on selecting only "newer" goals for execution is described and an algorithm is proposed for efficiently maintaining and determining precedence relationships and variable ages among parallel goals. Section 5 finally offers some conclusions.

2 Precedence and Memory Management in Sequential Systems

Sequential logic programming systems obtain much of their performance from doing their own stack-based memory management. Figure 1 shows a *very simplified* memory management model for a typical stack-based Prolog implementation. Although a realistic model, such as the Warren Abstract Machine (**WAM**) [20], includes several stacks (for "environments", "choice points", local and global data, "trailed" variables, etc.), the storage model will be reduced for the purposes of this discussion to a single stack. Each invocation of a goal allocates its local and global storage from the top of this stack. Depth-first execution of the set of rules listed in figure 1 leaves in this single stack the "trace" shown in figure 1-A. Note that *Choice Point* (CP) markers are left at points where alternatives are available which can be returned to during backtracking (in figure 1 it is assumed that **b** is the only predicate with "alternatives" at run time).

During forward execution this *single* stack simply grows with each goal invocation until a final success or a failure occurs, or until memory space is exhausted. In this last case, garbage collection is necessary in order to continue. However, memory space can be recovered during *backtracking*. For example, if **e** fails (within $\mathbf{b}_1$), the next alternative, $\mathbf{b}_2$, will have to be considered, and all storage involved in the computation of $\mathbf{b}_1$ can be discarded. This is done in all practical implementations by trimming the now invalid top portions of all

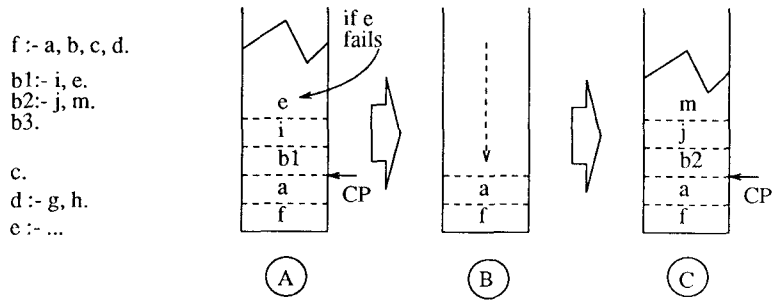


Figure 1: A Simplified Memory Management Model

stacks, as shown in figure 1-B. Certain bindings are normally undone while "unwinding" a portion of the "Trail" (a special stack which, for simplicity, is not shown in the model described herein). Forward execution can then proceed with $\mathbf{b}_2$ reusing the storage previously consumed by $\mathbf{b}_1$, as shown in figure 1-C. Despite its simplicity, this model illustrates the two main characteristics of space recovery on backtracking in sequential systems:

- *Complete Retrieval*: all space used in the computation of the previous alternative is recovered.

- *Storage availability for next alternative*: the storage is easily reused by the next alternative, since it is recovered from the *top* of the stack.

Note that the basic condition which makes space recovery possible is that at every point in the computation, *newer* structures are always stacked on top of *older* structures. The same ordering of newer over older structures that makes recovery of storage on backtracking possible is also essential in minimizing the number of values which need to be saved in the Trail and in the efficient implementation of last call (tail recursion) optimization [19].

The concept of goal age introduced above creates a *relation of precedence* (partial order) among goals. In practice, this relation of precedence is defined by the particular control strategy being used, i.e., for Prolog "older" means "closer to the root" and "to the left of" in the depth first, left to right execution tree. In order to make the discussion independent of any particular control strategy, a goal invocation **a** is herein defined as being "older" than another goal invocation **b**, represented as $\mathbf{a} < \mathbf{b}$, if, for a given control strategy, *all alternative solutions of* **b** *are to be tried before a new solution of* **a** *is attempted*. The results presented in this paper, although described in terms of AND-parallel systems which support "don't-know" nondeterminism, will be applicable to any execution model for which such a relation of precedence can be defined. Many parallel and coroutining models fall within this class.

Note that, because it explicitly represents the partial order, the simple storage

management model presented makes it easy to discuss compliance with the precedence conditions without the additional complexity of a more realistic scheme including registers, heaps, trails, etc. It has been shown [10] how other "popular" storage related optimizations (such as the recovery of local storage upon exit from a procedure, environment trimming, etc.) can be supported in a parallel system independently of the *scheduling strategy* being used. Therefore, these optimizations have also been left out of the storage model presented herein.

3 Towards Parallelism

One approach to memory management in parallel systems is to dynamically "allocate" a block of memory from a general pool for each of the goals to be executed in parallel. However, this approach in general relies on the existence of an underlying (operating system level) memory management system which will take care of such allocation and the maintenance of free space availability tables. Therefore, the performance of the parallel logic system will be limited by that of the underlying memory manager. As described in section 2, sequential systems obtain much of their performance from doing their own stack-based memory management. The purpose of the rest of this paper is to show how these optimizations can be efficiently extended to parallel systems. In the next section the single stack model will be extended to support AND-Parallel execution on multiple stacks. The discussion is presented in terms of a shared memory system. However, it applies just as well to distributed systems.

3.1 A Simplified, Multiple-Stack Model

An AND-Parallel logic program can in general be considered to comprise a series of sequential sections which eventually arrive at points where several execution paths can be taken simultaneously ("forks"). For the purposes of this discussion, it will be assumed that the control of this "forking" behavior is determined by annotations, in particular by Conditional Graph Expressions (**CGEs**) in a Goal Independence AND-Parallel model [11]. As an example, consider the following annotation for the **f** clause of figure 1 (variable names have been added to make the annotation meaningful):

```
f(X,Y,Z) :- a(X,Y), (indep(X,Y) | b(X) & c(Y,Z)), d(X,Y,Z).
```

The presence of the **CGE** "`(indep(X,Y) | b(X) & c(Y,Z))`" determines that during the execution of **f**, **a** has to be executed first, and then **b** and **c** can be executed in parallel if `X` and `Y` are determined to be independent at run-time. **d** will have to wait for all of them to finish prior to its execution.

A possible execution of the parallel AND-"branch" described above in a simplified multiple-stack model is represented in figure 2-A. Note that each processor manages a **physical stack** (the part of common memory

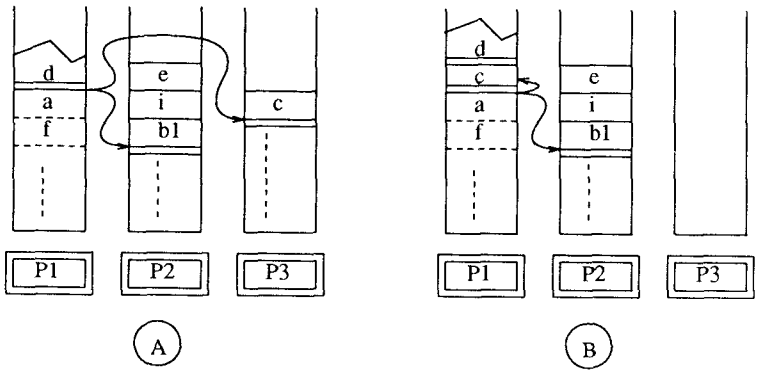


Figure 2: Parallel, Multiple-Stack Execution

corresponding to it) on which several **stack sections** can be allocated[1]. In this case, execution of **f** starts in *P1*'s (physical) stack but, as **a** succeeds, goals **b** and **c** are executed remotely and in parallel in *P2* and *P3*. All new data and control structures created by the execution of goals **b** and **c** are located within their respective stacks. When **b** and **c** finally succeed, execution of **d** can continue in *P1*, as shown in figure 2-A. The single stack of the sequential model is now distributed across the stacks corresponding to different processors.

An alternative execution of the clauses in the example above is shown in figure 2-B. In this case, after the success of **a**, execution of **b** starts as before in *P2* but now, since *P1* is idle (execution of **d** has to wait for **b** and **c** to succeed), it starts executing **c** itself, thus leaving *P3* free to perform some other task. When both **b** and **c** have succeeded, execution of **d** continues in *P1*. Note that if the several stack sections involved are combined in the right order into a single stack, this stack would be equivalent to the stack in figure 1-A.

As in the sequential model, this simplified multiple-stack model only reflects the relative *precedence* of objects in the different stacks, but it will be instrumental in showing in a simple way the factors which affect this precedence. The main such factor in a parallel environment is the *goal scheduling strategy*. In the next section a general *distributed goal scheduling strategy* will be presented. The following sections will first describe the problems posed by simply combining the general memory management and goal scheduling strategies presented in parallel backtracking systems and then propose a *unified scheduling and stack-based memory management scheme* capable of avoiding such problems.

---

[1]At this point it is assumed that there is only one process per processor: goals are distributed around the system as **units of work** (rather than processes) to these single processes. This approach has been shown to significantly reduce overhead in a parallel system [10]. The one process per processor assumption will be revisited in section 4.3.

3.2 A Goal Scheduling Strategy

A possible goal scheduling strategy is to have the processor which encounters goals which can be executed in parallel look for idle processors, assign one of these goals to each of the idle processors, and continue executing one of the remaining ones itself. The problem with this scheme is that in it all the "scheduling duties" (looking for idle processors, sending goals, etc.) are performed by a single processor which already has work to do, and thus the time involved in performing them adds up as sequential overhead. It is in general a better idea to put this burden in the hands of otherwise idle processors. The following is a distributed "load balancing" scheduling strategy in which idle processors pick up (i.e. "steal") goals from busy processors [10]:

- Each processor owns a *Goal Stack*, which is initially empty.
- All processors are initially idle. The user query is placed in the *Goal Stack* belonging to one of the processors. Execution starts in this processor and with this goal.
- When a processor arrives at a point where several goals are available for parallel execution those goals are pushed on to this processor's *Goal Stack*[2].
- Goals can be picked up from any *Goal Stack* for execution by the owner of the stack or by any idle processor. Idle processors pick up goals from other processors by looking into other processors' *Goal Stacks* until a non-empty *Goal Stack* is found. The topmost goal is then picked up from that *Goal Stack* and execution starts on it.
- When the execution of a goal finishes, the result (success or failure) is reported to the processor from which the goal was picked up (the "parent" processor).

The first processor thus starts working on the first goal and as it pushes goals on to its *Goal Stack* they may be picked up by other processors. These goals may in turn generate other parallel goals to be picked up by other processors and thus work spreads itself naturally as it becomes available. Note that this algorithm is valid even if there is only *one* processor present (or not faulty) in the system: since this processor can also pick up goals from its own *Goal Stack*, all goals scheduled for parallel execution will eventually get executed by this single processor. Note that if the "left bias" of sequential systems is to be preserved, goals actually have to be pushed on to the *Goal Stack* in *reverse order* (with respect to their sequence in the clause) [10].

Other similar distributed load balancing schemes have been proposed by Keller et al. [13], Burton and Sleep [3], and others. The main problem with "polling"

---

[2]Of course, a processor will in general keep one of the parallel goals for local execution thus saving a push-pop sequence in the *Goal Stack*.

schemes is that when the number of processors is large, moving from one *Goal Stack* (or "work queue") to another in order to find available work can be very inefficient. A global scheduling mechanism, capable of efficiently giving idle processors pointers to available work is desirable, but care must be taken to prevent this mechanism from becoming a serial bottleneck in the system. This is often the case if a single, global system work queue is used. An alternative hardware mechanism for "global" scheduling support is presented below.

3.3 A More Efficient Goal Scheduling Strategy

Consider the addition of a new architectural element to a parallel system: a "scheduling network". This element is connected to all processors, and it acts as a global scheduling mechanism. Its operation can be summarized as follows:

- Each processor continuously feeds a **value** into the network which represents its load.

- At any point in time any processor can ask the network for the Id. of the processor feeding in the highest **value**, and this Id. will be provided by the network with little delay.

Such a scheduling network can be implemented by anything from a wired-or bus (which would provide one maximum value) to a tree structure (such as a sorting network [1], which would provide the N maximum values), with different delays and cost depending on its complexity[3]. In particular, the delay and cost can be kept sublinear (logarithmic, at worst) with respect to the number of nodes (processors) involved. The same scheduling algorithm of the previous section can make use of such a network by simply modifying the way in which idle processors look for work:

- The number of goals in each processor's *Goal Stack* is continuously fed to the scheduling network[4].

- An idle processor thus receives from the scheduling network the Id. of the processor with the highest number of goals in its private *Goal Stack*. It then picks up a goal from that *Goal Stack* and starts working on it.

The potential scheduling bottleneck is now the (sorting) network itself, but its

---

[3]Note that, specially in a shared memory system, this function can be *emulated* using a globally accessed value in common memory. However, it is argued that the availability of a hardware mechanism such as that proposed can significantly reduce scheduling overhead.

[4]If multiple processes per processor are implemented, then the number of processes being run should also be used. These two numbers (available goals and processes currently being run on the processor) can be combined to provide a total **load** number which is fed to the network. The scheduling scheme would then ensure that idle processors always picked up work from the most heavily loaded processors.

action is limited to a very simple operation whose delay can be kept minimal. Since more than one processor could attempt to pick up a given goal simultaneously, *Goal Stacks* obviously have to be locked during this operation. In order to prevent many processors from fighting for access to the *Goal Stack* with the maximum number of goals, a simple optimization can be introduced: while the *Goal Stack* of a processor is locked, the value fed to the scheduling network will be zero. Thus, this processor will not receive requests from others until it is free again.

4 Relating Memory Management and Scheduling

The multiple-stack memory management scheme introduced in section 3 can be combined with the above described scheduling algorithms as shown in figure 3, where both the single stack of section 3 and the *Goal Stack* introduced above are being represented. The contents of the stacks represent a possible execution of the same example used in figure 1. Note how in figure 3-B goals **b** and **c** (which are available for parallel execution) are pushed on to the *Goal Stack* of the processor which encounters them (*P1*). From there they are picked up by free processors (figure 3-C) which work on them until they completely succeed. If these processors in turn generated new goals for execution in parallel, those goals would be pushed on to *their Goal Stacks* and picked up from there by other free processors or by themselves. In principle, after execution of a given goal is completed, a new one can be picked up and execution can proceed stacking all new data structures above the old ones[5] (figure 3-D,E,F, assuming the annotation "`d :- (true | g & h)`" for the **d** clause in figure 1). Special stack frames called "*Markers*" [10] (represented by a double horizontal line in figures 2 and 3) can be used for separating different **stack sections**, each one of them containing all structures related to the execution of a given "picked up" goal.

The goal scheduling and memory management schemes described thus far can be used directly in "committed choice" systems. These include all stack-based implementations of functional languages and of logic languages which make use of "don't care" non-determinism, such as Concurrent Prolog [17], Parlog [8], and GHC [18]. Execution in these models will proceed as in figure 3, allocating all structures corresponding to the execution of new goals on top of those corresponding to previous ones. The stack in this case simply becomes a "Heap". In the event of memory exhaustion, a (distributed) garbage collection algorithm is used in order to retrieve unused space [12]. Backtracking systems, on the other hand, can in principle *recover* storage during backtracking. This is not possible, however, unless goal scheduling and memory management are tied together, as shown in the next section.

---

[5] See section 4.1 for a more detailed discussion on this subject.

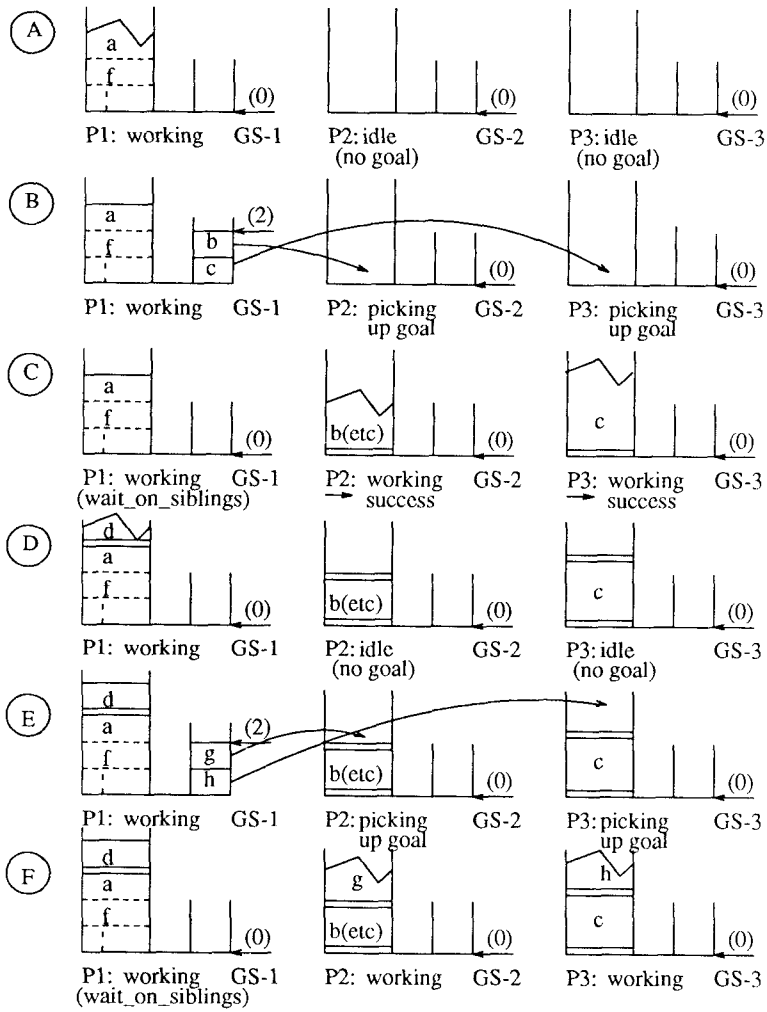


Figure 3: *Goal Stack* Based Goal Scheduling

4.1 Memory Management Problems Associated with Distributed Backtracking

If the memory efficiency of sequential systems is to be retained in a parallel implementation, equivalent conditions to those met sequentially (i.e. that at every point in the computation, *newer* structures always be stacked on top of *older* structures) are to be applied to the multiple stacks involved in the parallel execution. In particular, the following two conditions are to be met [9]:

1. The same precedence as in the sequential model should be maintained within each *stack section* during the execution of each particular goal.

2. If a new stack section is to be allocated on a physical stack containing other sections (as in figure 3-F, for goals **b** and **g**) then, if **b** is the last goal of the topmost section on the stack and **g** is the first goal of the new section, the relationship **b** $<$ **g** has to hold between goals **b** and **g** (i.e. **g** has to be "newer" than **b**).

Similar conditions have also been independently proposed by Borgwardt [2]. These two conditions, added to backtracking algorithms such as those proposed in [11, 10] ensure that, in the event of failure, backtracking in parallel systems will also be able to retrieve all stack space used in the computation of the failed alternative from the *top* of all stacks, and that execution of the new alternative will start from the new tops, so that there is free space available above for execution to continue.

It should be clear that, once a goal has started execution at any given processor, the *first* condition above, i.e. that newer structures always be stacked on top of older structures during the execution of that goal within a stack section, can be met simply by using the same techniques currently applied in conventional sequential models. Meeting the second condition above is more involved. The special problems associated with *not* meeting this condition will be illustrated in the following paragraphs[6].

4.1.1 The "Garbage Slot" Problem

Consider the situation in which a processor, having succeeded in the execution of a given goal, starts looking for another goal to work on. This is the case, for example, of processor 2 in figure 3-D,E,F: after the success of **b**, a new goal **g** is picked up. In order to illustrate a situation in which problems can arise, suppose that instead of **g**, a different goal **k** (from the execution of another parallel call somewhere else in the system) had been "picked up" and that **b** $<$ **k** does not hold (i.e. **k** is not "newer" than **b**). In this case, when dealing with a failure at some point during the subsequent execution of the program *it is possible that* **b** *may have to be backtracked before* **k**. If, as a result of such backtracking, **b** needs to be deallocated this represents only a minor problem: deallocation of **b** is possible, leaving an empty slot of "garbage" in the stack, and execution can continue (figure 4-A: the **"garbage slot"** problem). Although this space may be retrieved later through backtracking (i.e. if **k** is deallocated before anything else is stacked above it) this cannot be guaranteed in general. Therefore, complete retrieval of used space during backtracking is not preserved.

---

[6] As mentioned before, additional areas in which proper stack ordering is essential are the minimization of "trailing" and simple support for tail recursion optimization. The algorithms proposed herein also extend these optimizations to parallel implementations, although the discussion will be limited for simplicity to space recovery issues.

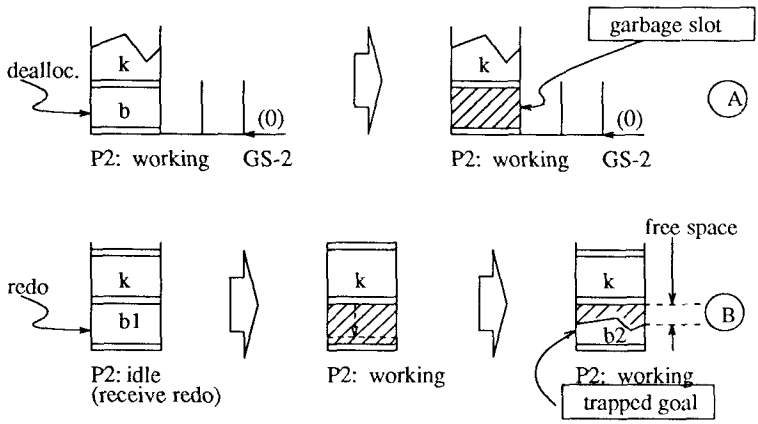


Figure 4: The "garbage slot" and "trapped goal" Problems

4.1.2 The "Trapped Goal" Problem

A more serious problem appears also during backtracking if, for the same situation depicted above, an alternative solution is needed from **b** (i.e. a *redo* message is sent to *P2* referring to **b**) and again **k** has not yet been deallocated. In this case part of the stack space occupied by **b** will be deallocated (down to the next "choice point" - figure 4-B) and a new alternative will be evaluated, its structures being stacked above this point. But note how, since there is no a priori limit on the storage which will be needed in the evaluation of this new alternative, the space available in the stack below **k** could be insufficient (figure 4-B, the **"trapped goal"** problem). If, on the other hand, **k** were actually "newer" than **b** (as is the case of **g** in figure 3-F), *all alternatives of* **k** *would have been tried, and* **k** *itself deallocated before* **b** *is ever required to backtrack.*

4.2 The Goal Restriction Solution

An effective approach towards ensuring that only "newer" goals are stacked above other goals is to *restrict the choice of goals which can be picked up by a given processor*: the scheduling algorithm is modified so that an idle processor *only picks goals which are "newer" than the topmost goal on its stack.*

Figure 5 shows the processor state diagram for a parallel backtracking system with *distributed goal scheduling restriction*. If after success or failure of a goal the stack is empty, clearly any goal in the system can be picked up for execution. This is, of course, always the case after initialization[7]. If, however, the stack still contains pending structures, the *goal restriction approach* dictates entering an *alternate idle loop*, where, rather than picking up any available goal

[7] In addition, if the "garbage slot" problem can be tolerated, any goal in the system can also be picked up if the underlying structures on the stack contain no alternatives. However, for the rest of this discussion it will be assumed that the "garbage slot" problem is to be avoided also.

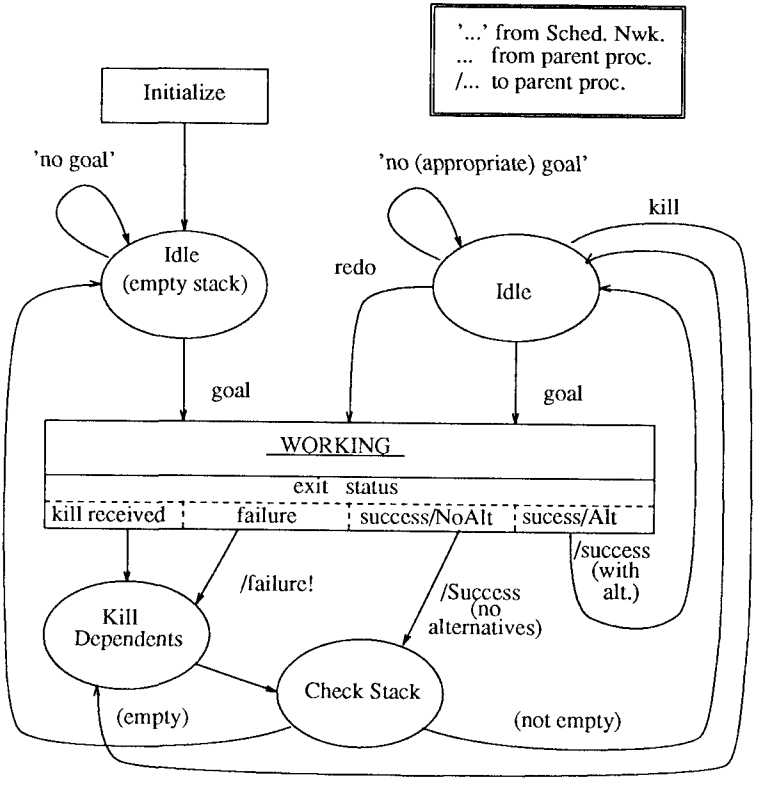


Figure 5: State Diagram for the Goal Restriction Approach

in the system, only an *appropriate* goal (defined as one which is "newer" than the last goal on the stack) will be looked for. When such a goal is found, work can continue on it, in confidence that its data structures can safely be grown on top of the old ones, since the relation of precedence ensures that the newer structures will always be deallocated before an underlying goal needs to be backtracked. Thus, the "garbage slot" and "trapped goal" problems are avoided.

One important advantage of this method is that, since the ordering of goals in the stack corresponds to its backtracking order, *any* **kill** *or* **redo** *message received from the parent necessarily refers to the* **last** *goal received.* This obviously means that all allocation and deallocation of structures on all stacks is always done to and from their tops. It also means that *kill* and *redo* messages do not have to identify the stack section they refer to: the last (topmost) goal executed in the processor receiving the message is implied.

Of course, some method has to be devised for efficiently determining precedence relationships among goals without having to traverse the whole

execution tree every time a goal has to be checked. Such a method is proposed in the following sections.

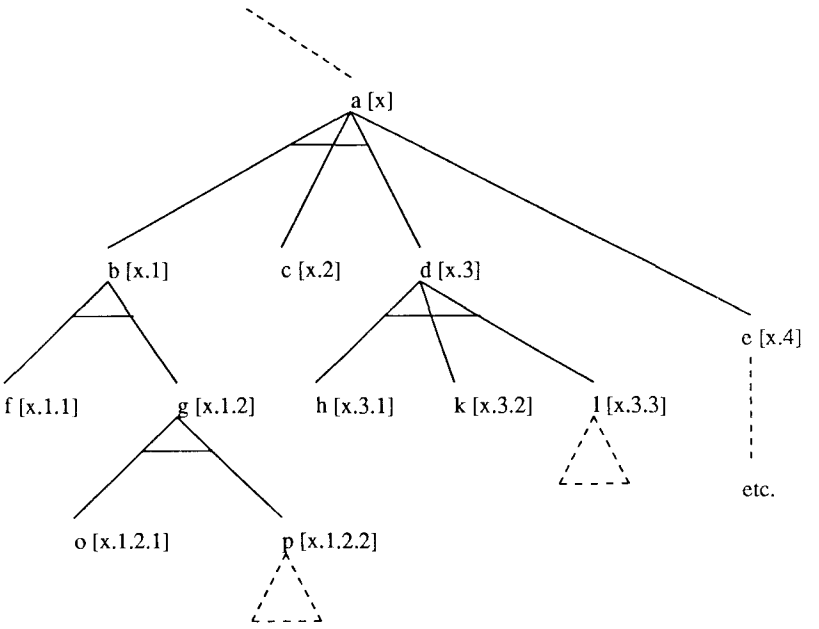
4.2.1 A Labeling Algorithm for Process Trees

Precedence relationships are maintained in sequential systems by simply associating an age with each object which is the physical address of this object in the stack. This scheme is clearly very efficient:

- The age of each object is implicit in its position (i.e. it doesn't have to be computed).
- Age comparisons between objects are reduced to address comparisons.

The same algorithm can be used in a multiple-stack model *within each stack section*. In addition, if the relation of precedence is maintained between the different sections on a single physical stack (as in the goal restriction solution proposed) then the same algorithm can also be used across stack sections within the same physical stack. If objects are located in different physical stacks, however, address comparisons are not meaningful. Ages have to be explicitly assigned to each section to make it possible to determine age relationships across different physical stacks.

Figure 6 represents an AND-Process tree for the set of clauses given therein. Note that each branch of the tree corresponds to sequential computation performed within a stack section. Branching points represent parallel calls and correspond to boundaries between sections. An algorithm is needed which always assigns a "greater" value (according to some comparison criteria) to a node than to any "older" node. For example, for Prolog semantics, a node **x** always has to be assigned a value greater than that of all nodes which would be visited before **x** in a depth-first, left-to-right traversal of the process tree. Note that since this tree grows dynamically at its leaves, a more sophisticated labeling algorithm than assigning simple integers to each node has to be implemented (i.e. in figure 6 new branches could be generated in the execution of **p** or **l**). An example of such a "dynamic" algorithm is also illustrated in figure 6: the age of a node is kept as a linked list of integers, the order of each element in the list representing the corresponding level in the tree, and the value of each element specifying the branches which would have to be taken in order to arrive at that node. Thus, referring again to figure 6, if */x/* is the age of node **a**, then */x.1.2.1/* is the age of node **o**, and */x.1.2.2/* that of node **p**. Is is easy to determine that **p** is "newer" than **o** by simply comparing the two lists. The general comparison algorithm to determine whether a goal **b** is "newer" than a goal **a** is then:



... a, (b & c & d), e.	p:- ...	k.
b:- (f & g).	l:- ...	o.
g:- (o & p).	c.	
d:- (h & k & l).	f.	
k:- (m & n).	h.	

Figure 6: An AND-Process Tree

Is $\mathbf{b} > \mathbf{a}$*?* Starting with the leftmost elements of both lists:

- If element($\mathbf{a}$) = element($\mathbf{b}$), select the next two elements and recurse.
- If element($\mathbf{a}$) < element($\mathbf{b}$) or no more elements in $\mathbf{a}$, answer **YES**.
- If element($\mathbf{a}$) > element($\mathbf{b}$) or no more elements in $\mathbf{b}$, answer **NO**.

There are at least two possible implementations for this scheme: the linked list can be kept implicitly by storing in each *Marker* the last element of the list and a pointer back to the *Marker* of the parent goal. When an age comparison is to be done, both chains of pointers are traversed until reaching the root, building the lists which represent the age of each node during the process, and then the

above algorithm is applied to the lists built. Although such an algorithm may appear excessively costly, note that the length of these lists is only proportional to the *depth* of the *process* tree, generally much smaller than that of the proof tree.

A more efficient approach (at some storage cost) is to always copy the parent's list (with the integer corresponding to the new node appended to it) to the stack section of the new node (for example, it can be stored in the Heap, with an entry in the corresponding *Marker* pointing to it). Then, both lists are always readily available for comparison. Also, a "lazy" combination of both of the approaches previously described offers an interesting compromise: a pointer to the parent is stored when the execution of a new stack section is started, but, if the chain ever has to be followed, then the explicit list built during this process is preserved in the stack section (i.e. in the Heap) with a pointer *to its head* stored in the *Marker*. Then, no subsequent age comparison needs to reconstruct this list and, at the same time, the construction and storage costs are avoided if an age comparison is never performed.

4.2.2 A Binary Labeling Algorithm for Process Trees

Although the algorithms presented in the previous section are probably workable, they are still much more expensive than the simple address comparison of sequential systems. However, a binary encoding of the above methods can afford very similar age comparison overhead to that found in sequential systems. One such binary encoding is to represent the integers in the list explicitly by enumeration, i.e. the "age" of each node in the previous section is encoded by representing "dots" as zeros and integers as strings of ones, where the length of each such string is the same as the value of the integer it represents. For example, referring again to figure 6, if x is the age of node **a**, then the age of node **k** ($/x.3.2/$) is represented as the string "$x0111011$", the age of node **p** as "$x01011011$", etc. In general, if the current age of a stack section is $\mathbf{S_0}=x$, upon arrival at a parallel call with n goals in it, the string $\mathbf{S_1}=\mathbf{S_0}01$ is assigned to the first goal and the string $\mathbf{S_i}=\mathbf{S_{i-1}}1$ to goal i, i=2,n. The leftmost zero of all codes (corresponding to the "root" goal) can be omitted.

Note that *if a decimal point is assumed to the left of these binary strings* (i.e. the strings are stored left-justified in a memory word, and the rest of the word is set to zeros) *the numerical value of the string representing a node is always greater than that of the string representing any "older" node.* This means that age comparisons can be reduced to simple arithmetic comparisons (i.e. without string traversal) in very much the same way as addresses are compared in a sequential implementation. Several levels in the process tree can be encoded within a single 32-bit word. If there are more levels, one bit (for example, the last one) can be reserved to indicate that the string is continued in the next memory word. The two first memory words of both ages are compared first. Subsequent words need only be compared if the preceding words are identical for both ages and both words being compared have "continuation bits". Again note that the length of the codes used depends only on the size of

the *process* tree, generally much smaller than that of the proof tree. In a work-based (rather than process-based) system [10] no processes are actually created when there are no idle processors in the system, thus further reducing the length of the codes. Also note that codes will be "recovered" during backtracking.

4.2.3 Goal Restriction Scheduling

A "polling" scheme can be used in conjunction with the above algorithms in order to support goal restriction scheduling. Idle processors look into other processors' *Goal Stacks* for goals which are "newer" than the last goal in their stack. It is also conceivable again to support this function in hardware through a scheduling network similar to that proposed in section 3.3. A similar scheduling algorithm can then also be used: a processor feeds in the "age" of the goal on top of its local stack to the network, and the network in turn returns the Id. of the processor having the "oldest" goal in its *Goal Stack* which is "newer" than the fed-in age. When there are several "eligible" goals, load is used as an additional factor.

4.3 Extending the Goal Restriction Approach: Creating Multiple Stacks (Processes) per Processor

Although the goal restriction approach as presented above makes very efficient use of its stack, it can sometimes leave processors idle despite the availability of work in the system (non-empty *Goal Stacks*) if none of such goals is "newer" than the last goal on this processor's stack. In each such case, for systems with a limited number of processors (i.e. where processor utilization is of great importance), a *new stack* can be created for such a processor and execution can then continue on this new stack with any goal in the system (of course, the oldest available goal should be chosen). A processor state diagram for such an approach is shown in figure 7. An idle processor will first look for *appropriate* goals in the system. As long as *appropriate* goals can be found, they are executed on the existing stack taking advantage of the inherent memory management efficiency of the single stack approach. If no *appropriate* goals can be found, but there is work available in the system (i.e. *non-appropriate* goals) then a new stack is created and work continued on it.

Note that in a real implementation, beyond the single-stack model used herein, the "creation of a stack" implies generating a complete new set of stacks containing one of each of the areas used in a conventional implementation (i.e. for a **WAM**-based system, a new Stack, Heap, Trail, etc.) and saving the machine state corresponding to the old stack (i.e. the values of all the registers). In fact, the operation which has been referred to as "creating a new stack" corresponds in more conventional terms to the creation of a new *process*. Thus, the fact that multiple stacks are supported per processor can be viewed as having several processes in a given processor (one for each "stack" in use), some of them "WORKING" and others waiting for a *kill* or *redo* message. Such a multi-stack approach is relatively straightforward to implement in a system with a large addressing space, and virtual memory support. It offers very good

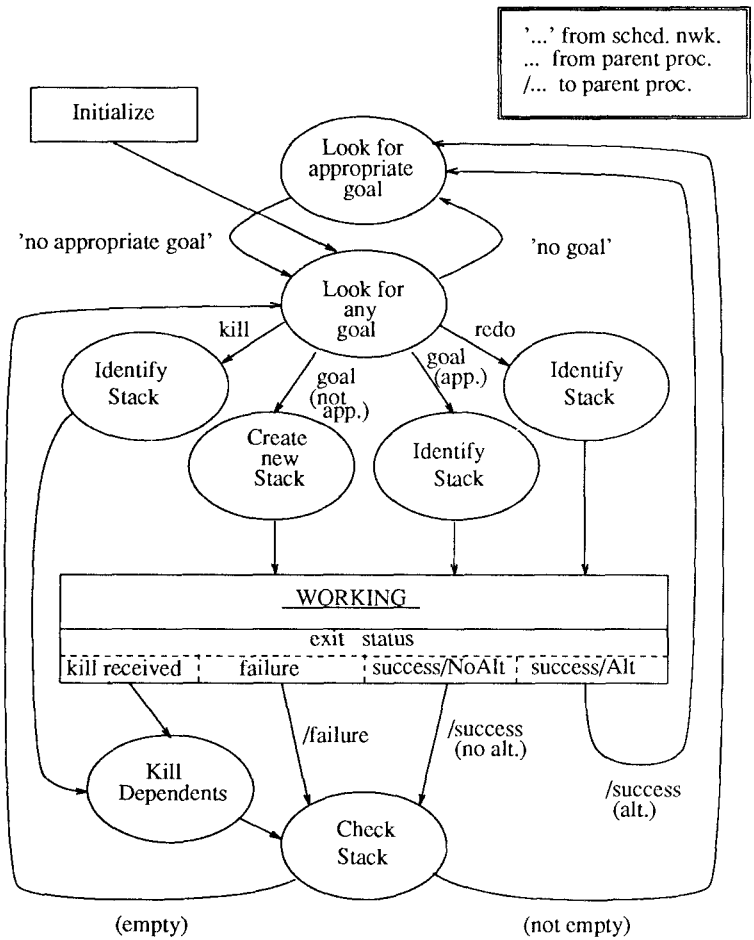


Figure 7: Creating Multiple Stacks per Processor

processor utilization, at the expense of stack (process) creation and switching overhead. Other stack management algorithms are presented in [10].

5 Conclusions

In the previous sections the interactions among goal scheduling, precedence, and memory management in parallel logic program implementation have been discussed. It has been shown how, for AND-parallel systems which support "don't know" non-determinism, special care has to be taken during goal scheduling if the space recovery characteristics of sequential systems are to be preserved. The goal restriction approach proposed ensures support for such

optimizations in parallel systems for which a partial ordering among goals can be defined. This includes a large class of parallel and coroutining systems. The memory management strategy proposed then represents an efficient alternative to "all heap" or "spaghetti stack" allocation. It is argued that the algorithms presented for the determination of relative age between parallel goals make the efficient implementation of such an approach possible. A multiple stack per processor approach was also proposed for increased processor utilization in systems with a small number of processors.

References

[1] K. E. Batcher.
Sorting Networks and Their Application.
AFIPS Conf. Proc. 32:307-314, 1968.

[2] P. Borgwardt and D. Rea.
Distributed Semi-intelligent Backtracking for a Stack-based AND-parallel Prolog.
In *Proceedings of the 1986 Symposium on Logic Programming*, pages 211-222. IEEE Computer Society, 1986.

[3] F. W. Burton and M. R. Sleep.
Executing Functional Programs on a Virtual Tree of Processors.
In *Functional Programming Languages and Computer Architecture*, pages 187-195. October, 1981.

[4] J.-H. Chang, A. M. Despain, and D. DeGroot.
AND-parallelism of Logic Programs Based on Static Data Dependency Analysis.
In *Digest of Papers of COMPCON Spring '85*, pages 218-225. 1985.

[5] A. Ciepilewski and S. Haridi.
Control of Activities in the Or-Parallel Token Machine.
In *1984 International Symposium on Logic Programming, Atlantic City*, pages 49-58. IEEE Computer Society Press, Silver Spring, MD, February, 1984.

[6] J. S. Conery.
The AND/OR Process Model for Parallel Interpretation of Logic Programs.
PhD thesis, The University of California at Irvine, 1983.
Technical Report 204.

[7] Doug DeGroot.
Restricted And-Parallelism.
Int'l Conf. on Fifth Generation Computer Systems , November, 1984.

[8] S. Gregory.
Design, Application and Implementation of a Parallel Logic Programming Language.
PhD thesis, Imperial College of Science and Technology, 1985.

[9] M. V. Hermenegildo.
An Abstract Machine for Restricted AND-parallel Execution of Logic Programs.
In *Proceedings of the Third International Conference on Logic Programming*, pages 25-40. Springer-Verlag, 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, August, 1986.

[11] M. V. Hermenegildo and R. I. Nasr.
Efficient Management of Backtracking in AND-parallelism.
In *Proceedings of the Third International Conference on Logic Programming*, pages 40-55. Springer-Verlag, 1986.

[12] A. Houri and E. Shapiro.
A Sequential Abstract Machine for Flat Concurrent Prolog.
Technical Report CS86-20, Dept. of Computer Science, The Weizmann Institute of Science, Rehovot 76100, Israel, July, 1986.

[13] R. M. Keller, F. C. H. Lin, and J. Tanaka.
Rediflow Multiprocessing.
In *Digest of Papers, Spring COMPCON '84*, pages 410-417. IEEE Computer Society, 1984.

[14] R. A. Kowalski.
Predicate Logic as a Programming Language.
Proc. IFIPS 74 , 1974.

[15] Y.-J. Lin, V. Kumar, and C. Leung.
An Intelligent Backtracking Algorithm for Parallel Execution of Logic Programs.
In *Proceedings of the Third International Conference on Logic Programming*, pages 55-69. Springer-Verlag, 1986.

[16] R. A. Overbeek, J. Gabriel, T. Lindholm, and E. L. Lusk.
Prolog on Multiprocessors.
Technical Report, Argonne National Laboratory, Argonne, Ill. 60439, 1985.

[17] E. Y. Shapiro.
A subset of Concurrent Prolog and its interpreter.
Technical Report TR-003, ICOT, January, 1983.
Tokyo.

[18] K. Ueda.
Guarded Horn Clauses.
Technical Report TR-103, ICOT, 1985.
Tokyo.

[19] D. H. D. Warren.
An improved Prolog implementation which optimises tail recursion.
DAI Research Report 141, University of Edinburgh, 1980.

[20] D. H. D. Warren.
An Abstract Prolog Instruction Set.
Technical Note 309, SRI International, AI Center, Computer Science and Technology Division, 1983.