An And/Or-Parallel Implementation of AKL

Johan Montelius
jm@sics.se

Khayri A. M. Ali
khayri@sics.se

Abstract

The Agents Kernel Language (AKL) is a general purpose concurrent constraint language. It combines the programming paradigms of search-oriented languages such as Prolog and process-oriented languages such as GHC.

The paper is focused on three essential issues in the parallel implementation of AKL for shared-memory multiprocessors: how to maintain multiple binding environments, how to represent the execution state and how to distribute work among workers.

A simple scheme is used for maintaining multiple binding environments. A worker will immediately see conditional bindings placed on variables, all workers will have a coherent view of the constraint stores. A locking scheme is used that entails little overhead for operations on local variables.

The goals in a guard are represented in a way that allows them to be inserted and removed without any locking. Continuations are used to represent sequences of untried goals. The representation keeps the granularity of work more coarse.

Available work is distributed among workers in such a way that hot-spots are avoided. And- and or-tasks are distributed and scheduled in a uniform way.

1 Introduction

The Agents Kernel Language (AKL) is a general purpose concurrent constraint language. It combines the programming paradigms of search-oriented languages such as Prolog and process-oriented languages such as GHC.

This paper presents the design of a parallel implementation developed at SICS. In a parallel implementation several problems have to be solved: how to handle the multiple binding environments, how to manipulate the execution state, how to distribute available work and how to manage storage and perform garbage collection.

The computational model of AKL gives two orthogonal reasons for maintaining multiple environments. One is due to the deep guards, the other to the

non-deterministic execution. In the implementation the multiple environments created by non-deterministic execution are handled by explicit copying of structures in the execution state. The binding scheme need therefore only handle multiple environments caused by the deep guards.

The binding environments should be represented in a way that induces as little overhead as possible but at the same time allows several processes to execute in parallel without too much interference. The perfect solution does of course not exist, any solution will have to make some sacrifices, the question is when the penalty should be taken.

The question of how the execution state should be represented and manipulated is not as critical as the binding scheme. The operations on the execution state are not as frequent as the operations on terms. The only critical part is the management of goals: new goals are constantly created and solved goals are removed, and these operations must be performed as efficiently as possible.

Distribution of work is an area in which little research has been done. Most work has been done for flat committed choice implementations and for or-parallel implementations of Prolog. These implementations need only consider one type of parallelism. The Andorra-I system is one of the few implementations where research on load balancing for both and- and or-parallel work has been conducted [8]. The solution in our implementation is as will be seen quite different.

Managing the storage heap and garbage collection has been presented in another paper [1].

Section 2 gives an overview of the computation model and execution state of AKL. An introduction to the computation model of AKL can be found in [9]. For a formal treatment, see [4]. Section 3 presents the binding scheme. It also includes an evaluation and justification of the scheme. Section 4 describes how the execution state is represented. Section 5 describes how available work is represented, scheduled and balanced among processors (workers). This work is still at an early stage so no evaluation of the scheduling procedures has been made. Section 6 is a summary of the paper.

2 The Computation Model

The AKL computation model is defined as a set of transformations of an execution state called the *configuration*. The operations on the configuration are defined by set of rewrite rules. There are four *determinate* rewrite rules (*try*, *pruning*, *promotion* and *failure*) and one *non-determinate* rewrite rule (*choice splitting*). We will first describe the structure of a program, then the structure of a configuration and the rewrite rules. Finally we will explain how a worker performs the operations on a configuration.

a program

An AKL program is a set of predicate definitions. Each predicate is defined by a set of guarded clauses consisting of a head, a guard, a guard operator and a body. The head is a program atom, the guard and body contain program atoms and constraint atoms. There are three types of guard operators: conditional, commit and wait. All clauses of a definition has the same type of guard operator.

the configuration

A goal is either a program atom, a constraint atom or a *choice-box*. A choice-box represents the execution of a program atom and contains a sequence of *guarded goals*. Each guarded goal represents a guarded clause and consist of a guard, a guard operator and a body. The guard is represented by an *and-box*. An and-box contains a sequence of goals, a set of constraints and a set of variables. The body is represented by a sequence of atoms. The root of the configuration is an and-box. We will in this paper informally refer to the children, siblings and parents of boxes.

configuration

$$
\begin{array}{rcl}
\langle \textit{and-box} \rangle & ::= & \mathbf{and}(\langle \textit{sequence of goals} \rangle)^{\langle \textit{set of constraints} \rangle}_{\langle \textit{set of variables} \rangle} \\
\langle \textit{choice-box} \rangle & ::= & \mathbf{choice}(\langle \textit{sequence of guarded goals} \rangle) \\
\langle \textit{guarded goal} \rangle & ::= & \langle \textit{guard} \rangle \langle \textit{guard operator} \rangle \langle \textit{body} \rangle \\
\langle \textit{guard} \rangle & ::= & \langle \textit{and-box} \rangle \\
\langle \textit{body} \rangle & ::= & \langle \textit{sequence of atoms} \rangle \\
\langle \textit{goal} \rangle & ::= & \langle \textit{choice-box} \rangle \mid \langle \textit{atom} \rangle \\
\langle \textit{atom} \rangle & ::= & \langle \textit{program atom} \rangle \mid \langle \textit{contraint atom} \rangle \\
\langle \textit{guard operator} \rangle & ::= & \rightarrow \mid \ ' \mid | \mid ? \quad \text{(conditional, commit, wait)}
\end{array}
$$

A variable is *local* to an and-box, referred to as the *home* of the variable. Variables that have their home in the path from the root to the and-box are *external* to the and-box. The sets of constraints in the configuration form a hierarchy of *constraint stores*. A constraint on a local variable is called an *unconditional constraint*. A constraint on an external variable is called a *conditional constraint*. Constraints are only visible in the subtree formed by the and-box in which the constraint resides.

An and-box is *solved* if the sequence of goals is empty. An and-box is *quiet* if all constraints on external variables are entailed by the constraint stores above the and-box. Since a guard is represented by an and-box we will also speak about solved and quiet guards and guarded goals. A solved and-box $\mathbf{and}()^{\theta}_{V}$ can be written as θ_V.

An and-box is *stable* if no determinate operation can be performed in the subtree formed by the and-box even if a satisfiable constraint on an external variable is added to the configuration. The notion of stability is central to the idea of combining search and concurrency.

rewrite rules

The rewrite rules are described as transformations of a component in a configuration. The letters R and S denote sequences of goals, A denotes an atom, B denotes a sequence of atoms, G denotes an and-box, E and F denote sequences of guarded goals, U and V denote sets of variables and θ and ϕ denote sets of constraints. The symbol % will be used instead of a guard operator.

try

A program atom is tried by replacing it by a choice-box with one guarded goal for each clause in the definition. Each guarded goal will have a guard, a guard operator and a body that correspond to the a guarded clause in the definition. The guard will have a set of local variables and a set of constraints that unifies the arguments of the program atom with the arguments of the head of the clause.

$$\mathbf{and}(R, A, S)^{\theta}_{V} \Rightarrow \mathbf{and}(R, \mathbf{choice}(G^{\phi}_{U} \% B, \ldots), S)^{\theta}_{V}$$

A constraint atom is tried by removing the atom and adding the corresponding constraint to the set of constraints.

$$\mathbf{and}(R, A, S)^{\theta}_{V} \Rightarrow \mathbf{and}(R, S)^{\theta \cup \{c\}}_{V}$$

pruning

A guarded goal that is solved and quiet may, depending on the guard operator, prune its siblings whereby the pruned guarded goals are removed from the configuration. The wait guard operator does not allow pruning. The commit guard operator allows pruning in both directions. The conditional guard operator only allows pruning of the siblings to the right.

$$\mathbf{choice}(E, \theta_V \mid B, F) \Rightarrow \mathbf{choice}(\theta_V \mid B)$$
$$\mathbf{choice}(E, \theta_V \rightarrow B, F) \Rightarrow \mathbf{choice}(E, \theta_V \rightarrow B)$$

promotion

If a choice-box has a single solved guarded goal as its only child the guarded goal may, depending on the guard operator, be promoted. A guarded goal is promoted by replacing the parent choice-box with the body of the guarded goal and adding the constraints and variables of the guard to the parent and-box.

The commit and conditional guard operators require that the guard is quiet. The wait guard operator does not place any extra requirements on the operation.

$$\mathbf{and}(R, \mathbf{choice}(\theta_V \% B), S)^{\phi}_{U} \Rightarrow \mathbf{and}(R, B, S)^{\phi \cup \theta}_{U \cup V}$$

failure

A choice-box with no guarded goals is removed and the false constraint is added to the constraint store of the and-box.

$$\mathbf{and}(R, \mathbf{choice}(), S)^{\theta}_{V} \Rightarrow \mathbf{and}(R, S)^{\theta \cup \{\bot\}}_{V}$$

If the set of constraints of an and-box is disentailed, by the constraint stores above the and-box, it fails. A guarded goal containing a failed guard is removed from the configuration.

$$\mathbf{choice}(E, G^{\theta}_{V} \% B, F) \Rightarrow \mathbf{choice}(E, F)$$

choice splitting

A *candidate* is a solved guarded goal with a wait guard operator. If an and-box is stable and the subtree formed by the and-box contains a candidate, a choice splitting operation can be performed.

$$\begin{aligned} &\mathbf{and}(R, \mathbf{choice}(\theta_V ? B, E), S)^{\phi}_{U} \Rightarrow \\ &\mathbf{and}(R, \mathbf{choice}(\theta_V ? B), S)^{\phi}_{U}, \mathbf{and}(R, \mathbf{choice}(E), S)^{\phi}_{U} \end{aligned}$$

The choice splitting rule will duplicate work (R and S), something that must be avoided until all other possibilities have been tried. The stability requirement prevents the split operation until it is known that no determinate operation can be performed in the and-box.

A stable and-box is in some sense in a deadlock i.e. no computation can proceed without additional information in form of new constraints. The choice splitting operation will as a result create an and-box with a choice box that is directly eligible for promotion. The promotion will hopefully add new constraints to the and-box making determinate operations possible. The deadlock situation is thus broken by the transformation. Since choice splitting and promotion are connected, the two operations are often referred to as "non-deterministic promotion".

In a naive implementation, stability detection can be avoided by treating the whole configuration as trivially stable when all determinate operations have been performed. This will of course sequentialise all split operations but this is not the biggest problem with such an approach. The biggest problem is that the root can not be treated as stable since it will interact with the outside world. Some goals in the root of the configuration will be connected to streams of information that in a reactive program can contain information that will add constraints to the configuration. Any implementation used for real reactive systems must therefore be able to detect stability correctly.

the worker

A worker is a process that performs the transformations of the configuration. It is always "positioned" in an and-box or choice-box in the configuration. The box in which the worker is positioned is called the *current* box.

The worker is driven by a set of *tasks*, where each task is associated with an and-box or choice-box in the configuration. The worker must handle all tasks in the current box before it is allowed to move up to the parent box. During the execution of a task it may generate new tasks, but all new tasks will be associated with boxes in the subtree formed by the current box.

By ensuring that a worker always performs all tasks in a box before it can move up, half of the problem of detecting stability has been solved. If a worker is about to leave an and-box we know that all determinate operations within the and-box have been performed. The remaining problem is to determine if a constraint added above the and-box can make a determinate operation possible.

In the following sections we will not talk about guarded goals. Since each guarded goal contains an and-box, we will talk only about and-boxes.

3 The Binding Scheme

The constraints are in this presentation limited to equality on rational trees. Constraints will be referred to as bindings. Several solutions to the problem of maintaining multiple binding environments have been published ([5] gives a good overview and comparison). These solutions are however designed for or-parallel systems where different bindings exist in different or-branches in an execution state.

In the implementation or-nodes are not used; alternative environments are created by explicitly copying all structures involved. The implementation need only handle bindings on different levels in the configuration. Since the levels in an AKL configuration normally are fewer than the levels of or-nodes in a Prolog execution, a very simple scheme can be used. The scheme is close to the computation model and places little overhead on the most frequent operations.

term representation

Terms are represented by tagged pointers. In this presentation it is sufficient to distinguish unconstrained variables (UVA), constrained variables (CVA), term references (REF), atoms (ATM) and structures (STR).

An unconstrained variable holds, apart from its tag, a reference to the home and-box. The reference is called *environment identifier* (env). A constrained variable holds a reference to a structure holding the environment identifier and a list of *suspensions*. All variables are created as unconstrained variables: only if a suspension is added to the variable will it turn into a constrained variable. The term reference is used for variable-variable bindings. The dereferencing

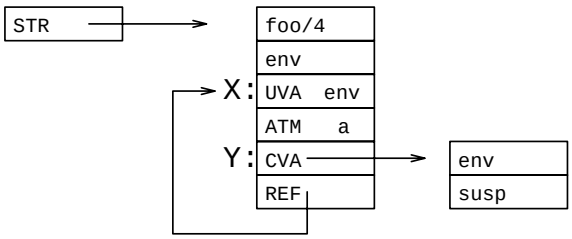

Figure 1: foo(X,a,Y,X)

procedure will follow any term references until a structure, an atom or variable is found.

Atoms hold only an identifier, typically a short integer or an index to an atom table. Structures hold, apart from the functor and its arguments, an environment identifier. Figure 1 shows an example of how the structure "foo(X,a,Y,X)" (where the variable "Y" is constrained) can be represented.

Note that the representation of variables is different from the representation of variables in WAM [16] where unbound variables are represented by a self-reference. This means that care must be taken when a variable is bound to another variable or copied to a register. In WAM the pointer is simply copied and automatically becomes a reference to the variable. In the representation described a new term reference has to be created. The self-reference is used, as will be described later, as a lock.

who is local?

A promotion of an and-box will mark it as promoted and insert a forward reference to its parent and-box. To check whether a variable is local, the environment identifier of the variable is compared with a pointer to the current and-box. If they are equal the term is local, otherwise the environment identifier is examined. If it refers to a promoted and-box the forward pointer is used for the comparison. This step is repeated; if a non-promoted and-box is reached the variable is external. This scheme is similar to the scheme designed in [11].

binding lists

Unconditional bindings can never be removed and can therefore be recorded in place, i.e. the value of a variable is permanently replaced by the binding. Conditional bindings must only be visible in and below the and-box in which the binding occurs and are therefore recorded in a *binding list* local to the and-box.

A binding list consists of an entry for each constrained external variable. An entry contains, apart from a reference to the variable, the binding of the

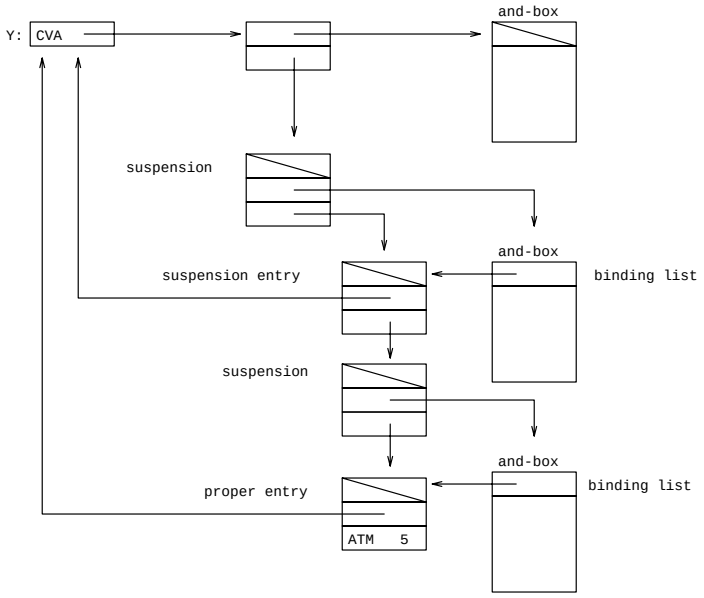

Figure 2: the hierarchy of suspensions

variable or a list of suspensions. An entry that contains a list of suspensions is called a *suspension entry*. An entry that contains a binding is a *proper entry*.

A suspension contains a reference to an and-box and a reference to an entry. A suspension on a constrained variable refers to an and-box immediately below the home of the variable. This and-box contains either a proper entry or a suspension entry for the variable. A suspension of a suspension entry refers to an and-box immediately below the and-box to which the entry belongs. The hierarchical structure of the suspensions is used both to maintain locality of suspensions and to detect stability. Figure 2 shows a constrained variable " Y" and the hierarchy of suspensions.

To find the current binding of an external variable each binding list from the current and-box to (but not including) the home of the variable must be examined. If no entry is found the variable is unbound. If an entry is found which binds the variable to another variable the binding of this variable must be found.

adding a binding

A local variable is bound to a structure or atom by replacing the value of the variable with the structure. A local variable is bound to another variable by replacing its value with a term reference to the other variable. If the bound

variable is constrained the suspended and-boxes are *woken*.

An external variable is bound to a structure or atom by first adding a proper entry to the local binding list. If the binding list contains a suspension entry for the variable this entry is reused and the suspended and-boxes woken. If no suspension entry was found in the current and-box the hierarchical structure of suspensions must be updated. Suspensions are added to each binding list from the parent and-box to (but not including) the home of the variable and to the variable itself. If the variable is an unconstrained variable it is turned into a constrained variable.

In a variable-variable binding where both variables are external, care must be taken. The *least external* (the variable whose home and-box is closer to the current and-box) is bound to the most external. If both variables belong to the same and-box either direction can be chosen but suspensions must be added for both variables.

waking an and-box

When an and-box is woken the worker knows which entry is to be re-examined and this entry is removed from the binding list. If the entry is a proper entry the unification is retried. If the entry is a suspension entry the suspended and-boxes are woken.

Notice that the hierarchical structure of suspensions gives perfect information about which and-boxes must be woken. It is not necessary to examine an and-box to determine if it is below the current and-box and thus is to be woken.

locking

The description above does not consider how the binding scheme works in a parallel implementation, where more than one worker can access and modify the representation. To guarantee a consistent state a locking scheme is needed. This section describes a locking scheme that allows several workers to add new bindings in parallel.

A variable is locked by atomically exchanging its current value with a reference to the variable itself. If a self reference is returned the locking operation is retried. The circular reference will lock the access to the variable. Any worker that tries to dereference the value will be stuck in a loop until the lock is released, so the locking scheme does not change the dereference procedure.

There are two possible hazards in the binding scheme: circular references and deadlock situations. If two local variables are unified the binding scheme does not specify the direction of the binding. Two workers might create a circular reference (one worker places a reference in "X" to the variable "Y" while another worker places a reference in "Y" to "X"). If two external variables of the same and-box are unified, suspensions should be added to both variables. Both variables must therefore be locked and a deadlock situation might occur.

To make the binding scheme deadlock free and to avoid circular bindings a well defined order of variables must exist. The order can be defined by the address of the variable[1].

Notice that by locking the variable itself no other worker can add an entry for the variable during the binding operation. This prevents two workers from simultaneously adding a conditional binding for the same variable. This is a drawback but can also be used as an advantage. When a worker has locked a variable it knows that no entries for this variable will be added to the binding lists. This means that the binding list can be searched for matching entries before new entries are added. A lock is of course needed to synchronize the manipulation of the binding lists. Each and-box has a lock that protects the list.

stability

Stability of an and-box is detected by examining the entries in the binding list. A *living entry* for a variable is either a proper entry for the variable or a suspension entry that refers to a *living* and-box with a living entry for the variable. An and-box is living unless it is marked as dead by a failure or pruning operation. An and-box is unstable if and only if it holds a living entry.

Observe that the stability check is driven by the suspension entries of the current and-box. There is no need to traverse the subtree to look for proper entries for external variables, since any such entry would reveal itself as a suspension entry in the and-box. The price for this is paid when external bindings are added but, as will be shown, the number of external bindings is very small.

A stable and-box is eligible for a choice splitting operation. The choice splitting is implemented by making a copy of the subtree formed by the candidate's parent and-box. The siblings of the candidate are not copied and the candidate is removed from the original subtree.

how do we copy?

When the subtree is copied new instances must be made of all variables that are local to the subtree. Variables that are external to the subtree should be shared by the two copies. The environment identifier of a structure is used to determine if the structure is local to the subtree. Structures that are external to the subtree cannot contain local variables and can be shared by both copies.

The copying procedure is a three-phase operation. In the first phase a copy of the tree is created but no terms are copied. A copied and-box is marked as copied and given a forward pointer to the copy. In the second phase terms are copied. If the environment identifier of a variable or structure refers to a

[1] The order need only be preserved during the lock operation, not during the whole execution.

marked and-box the term is local to the subtree and should also be copied. The third phase will reset all marked and-boxes and forward pointers.

Forward pointers can be used in local structures to enable copying of circular structures and avoid duplicating shared structures. The local structures are not accessible to any other worker and can be modified freely.

A copy operation can be performed by a worker independently of the execution in other parts of the configuration. No external structures are modified during the operation.

Ground terms need not be copied. As soon as it is determined that a structure is ground (during copying or garbage collection) a null identifier can replace the original identifier. In subsequent copy operations the ground term can be shared.

The overhead of creating the environment identifier is large for smaller structures such as lists cells but the environment identifier in a structure is not strictly necessary. External variables and ground structures can be shared and a structure need only be copied if any of its components needs to be copied. However, the existence of the environment identifier makes the detection of external and ground structures more efficient.

evaluation

The main advantage of the described binding scheme is that a worker can move freely in the configuration since it does not need to update any private information. This allows for fast task switching. The explicit representation of the constraint stores also makes bindings immediately visible to all workers; all workers have a consistent view of the bindings in the configuration.

A disadvantage is the non-constant time operations: to access or to add a new binding can in the worst case be a very costly operation. In practice it does not cause any problem. The majority of variable accesses are made to local variables, in which case the binding is found in place, or to variables that are local to the parent and-box, in which case only one binding list is examined.

To evaluate the binding scheme statistics were gathered from five programs. The programs were executed in a prototype implementation running with one worker. The programs are:

compiler The compiler compiling itself (about 2000 lines of AKL code). The program uses concurrency and deep guards. The deep guards are used for convenience, the program could be rewritten with only flat guards.

waves Waves in a torus (5 generations, dimension 9), a flat program (originally written in Strand by Ian Foster, translated to KL1 by Evan Tick).

life The game of life (10 generations in a 40 times 40 toroid), a flat program written in AKL. Each cell is implemented as a process that depends on all its eight neighbours.

Program	local	one level	two levels
compiler	97%	3%	0%
waves	65%	35%	0%
life	77%	23%	0%
scanner	87%	12%	1%
knights	81%	18%	1%

Figure 3: local vs. external bindings

program	one level	two levels
compiler	100%	-
waves	100%	-
life	100%	-
scanner	100%	0%
knights	94%	6%

Figure 4: access to external variables

scanner Find a pattern in a grid given x-ray information from rows, columns and diagonals. A program that uses both concurrency and non-determinism.

knights The knights tour (first solution on an 8 times 8 board), a program that uses both concurrency and non-determinism. It is the only program that uses more than three levels of and-boxes.

The programs can be divided into three categories. The **compiler** is a real life program i.e. not only written for benchmark purposes. The **waves** and **life** programs are both flat committed choice programs, and are interesting in that neither can be executed depth-first without any suspensions. Almost all of the traditional benchmarks for flat committed choice languages can be executed depth-first without any suspensions. Such programs, although they represent a big group, were not included since a single worker execution will not use any deep bindings. The **scanner** and **knights** are programs that utilise both concurrency and non-determinism, AKL's crowning glory.

Figure 3 shows the percentage of local and external bindings made. As is clearly seen the majority of bindings are made to local variables. The external bindings are almost always made to variables that belong to a parent and-box. Bindings that span two levels are rare.

Once an external binding has been made it can of course be accessed many times. If it is accessed in the same and-box in which the binding is made only

program	0	1	2	3	4	5
compiler	90%	95%	100%	-	-	-
waves	94%	100%	-	-	-	-
life	42%	69%	83%	96%	99%	99%
scanner	98%	98%	98%	99%	99%	99%
knights	42%	75%	98%	99%	100%	100%

Figure 5: accumulating percentage of the number of elements traversed in each binding list

one binding list is searched, only if the access is even further down the tree need more than one binding list be traversed. Figure 4 shows how many levels have to be searched before a binding is found or the home and-box is reached. As is clearly seen almost all accesses to external variables only traverse one binding list.

One may also wonder if a list of bindings is efficient enough and if it would not be more efficient to use a hash table. In Figure 5 the number of elements traversed in each binding list for each variable access is shown. The number of elements is as one can see very small. The need for a more efficient representation is not imminent.

The use of the environment identifier to determine if a variable is local or external is another possible hazard. To dereference the environment is a non-constant time operation but it turns out that this is a very infrequent operation. In the tests described above only two thousand, out of two and a half million ($< 0.1\%$), environments had to be dereferenced, in all other cases the environment identifier pointed directly to a non-promoted and-box.

related work

Gopal Gupta and Bharat Jayarama [5] describe criteria for or-parallel execution models of logic programs. Comparing the proposed scheme with or-parallel models is however not straightforward. The implementation of choice splitting sacrifices constant-time task creation whereas the binding scheme sacrifices constant-time variable access. The criterion in the development of the scheme has been that local variable access should be a constant-time operation. In order to achieve this, constant-time choice splitting or constant-time task switching had to be sacrificed. A copying strategy for choice splitting was chosen since it simplifies the binding scheme.

Binding schemes for Concurrent Prolog have, as in AKL, to deal with multiple levels of bindings and suspension of goals. The "deep scheme" proposed by Sato, H. et al. [6] is very similar to the binding scheme we present. That scheme also uses a hierarchy of binding lists that have to be searched for each

access of an external variable.

The ParAKL implementation of AKL [14] uses a binding scheme based on the PEPSys hashing scheme. A hierarchical structure of hash tables is used to represent the constraint stores. The hierarchical structure reflects how work has been spawned and not the structure of the configuration. If two workers are executing in the same and-box the last spawned will record its bindings in a private hash window. Research is being done on compile time analysis for stability detection [12].

The sequential prototype implementation of AKL [10] uses a trailing scheme to implement the constraint stores. All bindings are made in place but external bindings are trailed. The implementation uses only a dirty bit to detect stable and-boxes with the consequence that once an and-box has become unstable it will be marked as unstable even if it later becomes stable.

Andreas Podelski and Peter van Roy have presented [13] an algorithm called "the beauty and beast" for tests of entailment and disentailment. It is an algorithm that avoids the $O(n^2)$ complexity (in the number of constraints) that is inherent in the scheme presented in this paper. The beauty and beast has not been implemented for a deep language and does not address the question of stability.

4 The Execution State

One way of implementing a concurrent language is to use goal stacking [3, 2] i.e. immediately create representations of the goals in a promoted body. The approach has several advantages in that it provides a uniform execution model and fast task creation. One disadvantage is however that unnecesssary work is performed if a goal fails. This is of course not a problem in a flat committed choice language where failure is treated as an exception but can be a problem in a deep language where failure of goals is a normal behaviour. Another problem is that a conjunction of goals (that presumably interact) is divided and distributed among workers.

In WAM [16] the "environment" is used to represent the remaining goals in a body. It is also used in and-parallel implementations of Prolog such as &-Prolog [7] and DDSWAM [15]. The method has proved very efficient in the sequential implementations of AKL [10] and we believe that it has additional advantages in a parallel implementation. Apart from the advantages of lazy creation of goals it can keep the granularity of work more coarse.

goals

The sequence of goals in the guard is represented by a single linked list of *insertion points*. Each insertion point is either *dead*, referring to a *choice-box* or

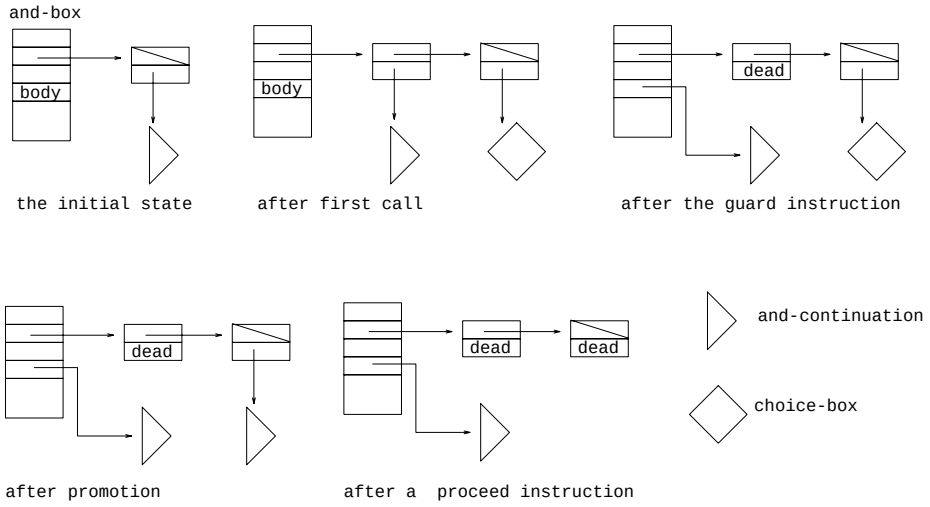

Figure 6: the execution of a guard

an *and-continuation*. To facilitate insertion the cells are linked right to left i.e. the leftmost goal in the guard is the last goal in the list.

An and-continuation contains the permanent registers, a code pointer and the insertion point of the continuation. The instructions of an and-continuation can be divided into three sections: instructions to create the goals of the guard, a guard instruction and instructions to create the goals of the body. An and-continuation is used to represent both untried goals in a guard and the body of an and-box.

insertion and promotion

When an and-box is first created, an and-continuation is created that represents both the guard and the body. When a goal is spawned from the and-continuation a new insertion point is inserted in the list and used for the created choice-box. When all goals of the guard have been spawned the guard instruction will be executed. The guard instruction will, if a promotion is not allowed, make the and-continuation the body of the and-box. The three first pictures in Figure 6 show the initial state of a guard execution, the state after the first and only goal has been spawned and the state after the guard instruction of the and-continuation has been executed.

When an and-box is promoted the body will inherit the insertion point of the parent choice-box. The last instruction of the body will mark the insertion point as dead (proceed instruction) or reuse it for the last goal (execute instruction). The last two pictures in Figure 6 show the state when the goal has been solved

and the and-continuation has been promoted, and the state after the execution of the proceed instruction of the promoted and-continuation.

This scheme allows the sequence of choice-boxes and and-continuations to be maintained without locking operations. The scheduling of available work guarantees that only one worker at a time has a reference to an and-continuation. The worker may manipulate it and its insertion point without any interference from other workers.

A disadvantage of the scheme is that the list of insertion points will contain dead entries. The list is, however, only traversed when all goals have been executed and it is to be decided whether the guard is solved or not. This operation is performed by the last worker to leave the and-box. Dead entries can be removed at any point in the execution, during execution or by garbage collection.

guards

A choice-box contains a reference to the insertion point of the choice-box, a *choice-continuation* and a single linked list of and-boxes. A choice-continuation represents untried guards and contains a code pointer and a copy of the argument registers. The and-boxes are, like the insertion cells, linked right to left.

The operation on the list of and-boxes is different from the operations on the list of goals. A worker must be able to remove (by pruning or failure) any and-box, which will require some locking. A fast insertion operation, provided in the list of goals, is not essential since new and-boxes are only created from a single choice-continuation and in the choice-split operation, both operations are infrequent. A lock in the choice-box will control all modifications on the list of and-boxes and the choice-continuation.

removing an and-box

When an and-box is removed, either by pruning, promotion or failure, workers must be prevented from entering the and-box and workers that are already in the and-box must be notified. To solve this each and-box keeps a bit vector that identifies workers in the subtree of the and-box. The bit vector is protected by a lock which must be taken before a worker is allowed to enter or leave the and-box. The worker also keeps a flag that indicates if the and-box is alive or dead.

When an and-box is removed the lock of the parent choice-box is taken. If the current and-box is still marked as alive (the and-box may have been pruned by another worker) the lock of the and-box that should be removed is taken. The locked and-box is marked as dead, the workers inside the and-box are signaled and the lock is released. The and-box can now be removed from the list of and-boxes. Finally the lock of the choice-box is released.

Any worker that enters the and-box will wait for the lock, and when the lock is taken the worker will detect that the and-box is dead and continue the execution elsewhere.

5 Scheduling

A worker will be driven by tasks. The worker will have a set of tasks for each level in the configuration. All tasks at a given level have to be handled before the worker is allowed to move up one level.

and-box tasks

There are two types of tasks associated with an and-box: *wake* and *a-cont*. Wake tasks are created when constrained variables are bound. Each suspended and-box will generate a wake task. The wake task contains a reference to the and-box and a reference to the entry that is to be re-examined. The a-cont tasks are created when a goal is called and when a body is promoted.

A worker that is executing the instructions of an and-continuation will when the call is made add an a-cont task to its stack and continue the execution of the called goal. Notice that this is the only task that refers to the and-continuation. Any worker that has this task can execute the remaining instructions without any locking operations. When a body is promoted the worker adds an a-cont task for the promoted and-continuation.

The last worker to leave the and-box will examine the and-box and, if allowed, promote the body of the and-box or prune sibling and-boxes.

choice-box tasks

There are three types of tasks associated with a choice-box: *c-cont*, *promote* and *split*. A c-cont task is created by a worker when there is more than one guard to explore. The c-cont task only indicates that a choice-continuation may exist, since it may have been removed by another worker in a pruning operation.

To understand how the promote and split tasks are created the choice split step must be understood. After the and-box has been copied the worker is positioned in the choice-box immediately above the and-box. The worker will move down to the new copy and perform a promotion of the copy of the candidate. There might however be things left in the original and-box. The original and-box might be stable, and be eligible for another choice-split operation, or contain a promotable and-box, since the candidate and-box has been removed. In either case (both are simple to detect) the worker must sooner or later return to the and-box. A split or promote task is added before the worker moves down in the copy. These tasks will be handled when the worker returns to the choice-box.

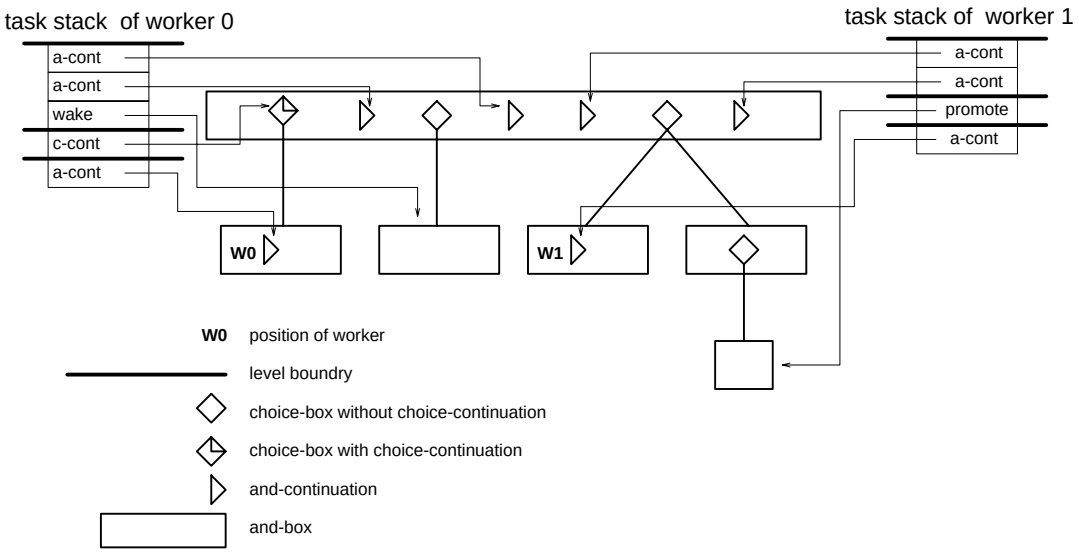

Figure 7: the stack of tasks for two workers

task stacks

The tasks are represented as references to structures in the configuration and can be stored in a stack private to the worker. Figure 7 shows a configuration with two workers. The configuration is shown schematically, the exact linking of the structures is not shown. Worker "0" has an a-cont task in its current and-box, a c-cont task in the parent choice-box and two a-cont tasks and a wake task in the uppermost and-box. Worker "1" has an a-cont task in its current and-box, a promote task in its parent choice-box and two a-cont tasks in the uppermost and-box.

The tasks are divided into segments, where each segment is associated with an and-box or choice-box in the current branch. The segments can be identified either from an extra stack or by inserting sentinels in the stack.

task distribution

All structures in the configuration are allocated in shared memory, so a task can be distributed simply by copying the task from the giving worker to the receiving worker. The tasks can be distributed among workers to dynamically balance the load. There is no distinction between and-box tasks and choice-box tasks: all tasks can be scheduled and distributed uniformly.

A worker will execute its own tasks until it reaches the root of the configuration and all tasks have been handled. It will then try to find another worker

with extra work and signal it to get some tasks. A worker can examine the stacks of other workers before it chooses which worker to interrupt. There are several approaches to how to select a worker:

- Interrupt the worker with a task as high up as possible.
- Interrupt the worker with the greatest number of tasks, take some of them.
- Prefer an a-cont task to a wake task.

The possibilities are many since the tasks are uniform and easily distributed. Any number of tasks can be taken, from one or several levels. It is also possible to stop at each level in the configuration to get more tasks from workers in the same branch.

6 Summary

We have described the fundamental components of a parallel implementation of AKL that allows both and- and or-parallel execution.

binding scheme

The problem of maintaining multiple environments is solved by two mechanisms: explicit copying to solve non-determinism and a binding scheme to deal with the hierarchy of constraint stores.

The explicit copying of the execution state causes a big overhead for non-deterministic programming. This approach was chosen in spite of the overhead since it does not impair deterministic executions. Non-determinism is a powerful programming concept, the price is paid only when it is used.

Two approaches exist when constructing the binding scheme for deep guards. One is to sacrifice constant-time variable access, the other is to sacrifice constant-time task switching. We have chosen to sacrifice constant-time variable access not because the constant-time task switching is more important but because the non-constant factor in variable accesses is controllable. All accesses to local variables are constant-time operations (modulo lock operation), and the nonconstant-time factor only appears when external variables are accessed. External variables are not introduced by the execution model but by the programmer. Again, when using a powerful concept you pay a price.

We have shown that external bindings are infrequent and when used in most cases only span one level. The evaluation was conducted by executing programs that we think, in this respect, represent the majority of programs. The evaluation supports the feasibility of the binding scheme.

The implementation is novel in that it keeps perfect track of stability, an issue that has been neglected in other designs. The information for stability detection is implicit in the binding scheme, no extra constructs or operations are

introduced. The stability information is essential in any implementation, one can not rely on the stability of the whole configuration. A parallel implementation would furthermore be impaired if stability detection is neglected since the choice splitting operations will be sequentialized.

execution state

The question of how the execution state should be represented and manipulated is not as critical as the binding scheme. The operations on the execution state are not as frequent as the operations on terms. The only critical part is the management of goals: new goals are constantly created and solved goals are removed, and these operations must be performed as efficiently as possible.

The goals in a guard are represented in a way that allows them to be inserted and removed without any locking. This is very important since this is the most frequent operation in the execution state.

Continuations are used to represent sequences of untried goals. The representation keeps the granularity of work more coarse. It will also avoid unnecessary work if a goal fails before all the goals have been spawned.

task distribution

A worker will be driven by tasks. The tasks are represented as references to structures in the configuration and can be stored in a stack private to the worker. All structures in the configuration are allocated in shared memory, so a task can be distributed simply by copying the task from the giving worker to the receiving worker.

The tasks can be distributed among workers to dynamically balance the load. There is no distinction between and- and or-parallel tasks: all tasks can be scheduled and distributed uniformly. This means that there is no need to separate an and-scheduler from an or-scheduler (as in [8]), one scheduler will handle all tasks in the system. This opens new possibilities for load balancing.

further investigation

A prototype implementation is under development and the preliminary results show that the binding scheme works well with little overhead compared to the sequential implementation.

The task distribution is only partly implemented. To experiment with different scheduling algorithms a graphical tool is being developed that will give a visual presentation of the execution given log messages from a real execution.

7 Acknowledgement

The research presented in this paper has been benefited from discussions with Seif Haridi, Sverker Janson, Takashi Chikayama and Gert Smolka, all whom we would like to thank. We would also like to thank Torkel Franzén who has helped us with the presentation.

The parallel implementation of AKL is developed as a part of the ACCLAIM Esprit project, EP 7195.

References

[1] Khayri A. M. Ali. A parallel copying garbage collection scheme for shared-memory multiprocessors. *New Generation Computing*, 14(3), August 1995.

[2] Vitor Santos Costa, David H. D. Warren, and Rong Yang. The Andorra-I engine: A parallel implementation of the Basic Andorra model. Technical note, University of Bristol, Department of Computer Science, March 1990.

[3] Jim Crammond. Implementation of committed choice logic languages on shared memory multiprocessors. Phd thesis, Heriot-Watt University, 1988.

[4] Torkel Franzén. Some formal aspects of AKL. SICS Research Report R94:10, Swedish Institute of Computer Science, 1994.

[5] Gopal Gupta and Bharat Jayaraman. On criteria for Or-Parallel execution models of logic programs. In Saumya Debray and Manuel Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 737–756, Austin, 1990. ALP, MIT Press.

[6] Sato H., Ichiyoshi N., Dasai T., Miyazaki T., and Takeuchi A. A sequential implementation of concurrent prolog - based on the deep binding scheme. In *The First National Conference of Japan Society for Software Science and Technology*, pages 299–302, 1984. In Japanese.

[7] M. V. Hermenegildo and K. J. Greene. &-Prolog and its performance: Exploiting independent And-Parallelism. In David H. D. Warren and Péter Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 253–268, Jerusalem, 1990. The MIT Press.

[8] de Catro Duatra Ines. Strategies for scheduling and- and or- work in parallel logic programming systems. In Maurice Bruynooghe, editor, *Proceedings of the 1994 International Logic Programming Symposium*, Ithaca, 1994. ALP, MIT Press.

[9] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra kernel language. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 167–186, San Diego, USA, 1991. The MIT Press.

[10] Sverker Janson and Johan Montelius. The design of the AKL/PS 0.0 prototype implementation of the Andorra Kernel Language. ESPRIT deliverable, EP 2471 (PEPMA), Swedish Institute of Computer Science, 1992.

[11] T. Miyazaki, A. Takeuchi, and T. Chikayama. A sequential implementation of concurrent Prolog based on the shallow binding scheme. In *Symposium on Logic Programming*, pages 110–118. IEEE Computer Society, Technical Committee on Computer Languages, The Computer Society Press, July 1985.

[12] Remco Moolenaar and Bart Demoen. Full parallel search in AKL. Submitted for publication.

[13] A. Podelski and P. Van Roy. The beauty and beast algorithm: Quasi-linear incremental tests of entailment and disentailment over trees. In Maurice Bruynooghe, editor, *Proceedings of the 1994 International Logic Programming Symposium*, pages 359–374, Ithaca, 1994. ALP, MIT Press.

[14] B. Demoen R. Moolenaar. A parallel implementation for akl. In *Lecture Notes in Computer Science, 714*, pages 246–261. Springer-Verlag, August 1993.

[15] Kish Shen. Implementing dynamic dependent And-Parallelism. In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 167–183, Budapest, Hungary, 1993. The MIT Press.

[16] D. H. D. Warren. An abstract prolog instruction set. Technical Report 309, SRI International, 1983.