The Penny Abstract Machine, an overview

Johan Montelius
jm@sics.se

Haruyasu Ueda
hal@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.

This paper gives an overview of the parallel implementation developed at SICS. The paper is focused on how the execution state is represented and how tasks are scheduled. A design is presented that induces very little overhead for locking operations, allows explicit deallocation of structures and provides a flexible task scheduler. The paper includes a preliminary evaluation of the system.

1 Introduction

The Agents Kernel Language (AKL) [5] 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. The implementation utilises both and-parallelism and or-parallelism. In a parallel implementation several problems have to be solved: how to handle the multiple binding environments, how to represent and manipulate the execution state, how to distribute available work and how to manage storage and perform garbage collection.

The binding scheme has been presented in [9]. The algorithm of the garbage collector of dynamic structures is presented in [1] and its parallel implementation is presented in [2]. This paper describes how the execution state is represented and how tasks are scheduled.

Section 2 gives an overview of the computation model of AKL. An introduction to the computation model of AKL can be found in [6]. For a formal treatment, see [4]. Section 3 gives an overview of the abstract machine for the parallel implementation. Section 4 gives some implementation details of the representation of the execution state. Section 5 is a description of how task are

scheduled. Section 6 gives an evaluation of the implementation. Section 7 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 a 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.

2.1 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 have the same type of guard operator.

2.2 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.

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* of the variable. 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 $\mathbf{and}()_V^\theta$ can be written as θ_V.

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

Figure 1: Configuration

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.

2.3 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, let it be c, 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.

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

failure

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

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

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

$$\mathbf{choice}(E, G_V^{\theta \cup \{\bot\}} \% B, F) \Rightarrow \mathbf{choice}(E, F)$$

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.

$$\mathbf{choice}(E_1, \mathbf{and}(R, \mathbf{choice}(\theta_V ? B, F), S)_U^{\phi}, E_2) \Rightarrow$$
$$\mathbf{choice}(E_1, \mathbf{and}(R, \mathbf{choice}(\theta_V ? B), S)_U^{\phi}, \mathbf{and}(R, \mathbf{choice}(F), S)_U^{\phi}, E_2)$$

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.

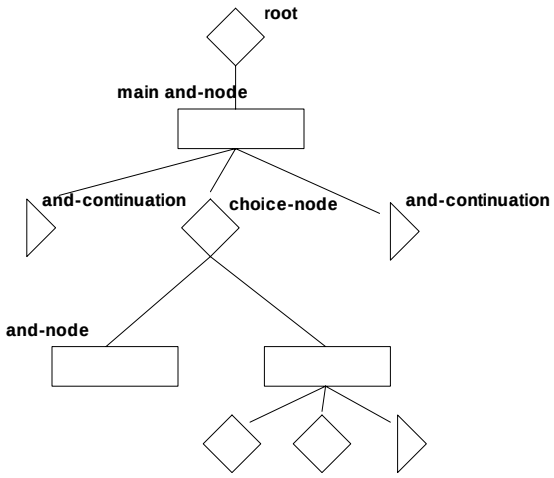

Figure 2: an execution state

3 The Abstract Machine

The *execution state* is a representation of the configuration. A *worker* is a process that performs the transformations on the execution state. The abstract machine is designed to allow multiple workers to collaborate in an execution state.

3.1 The execution state

The execution state is a tree of *choice-nodes*, *and-nodes* and *and-continuations*. The root of the tree is a dummy choice-node that serves only as a sentinel in the execution. The root node holds a single and-node that is referred to as the *main and-node*. Figure 2 shows a schematic execution state.

The description given in this section is as much as possible free of implementation dependent details. Implementation details will be described in the following sections.

and-node

An and-node is the representation of a guarded goal. It contains: a list of choice-nodes and and-continuations that represents the sequence of goals in the guard, an and-continuation that represents the body, a reference to its parent choice-node, a type, the identifiers of all workers installed in the and-node, a lock, a *forward pointer* and a set of *entries*. The type of the and-node is used to represent the guard operator of the guarded goal but it is also, as will be

shown in section 3.3.3, used for other purposes. The forward pointer is used if the body of the and-node is promoted.

Entries are mainly used for representing constraints on external variables but are also used to keep track of suspended computations. An entry is either a *unifier entry*, a *suspension entry*, *continuation entry* or a *dead entry*. All entries hold a reference to the and-node to which they belong. A unifier entry holds a reference to a variable and a term which the variable is constrained to be equal to. A suspension entry holds a reference to a variable and a set of *suspensions*. A continuation entry holds only a possibly empty set of suspensions.

Suspensions are referenced both from suspension entries, continuation entries and constrained variables. A suspension carries a reference to either an and-continuation, a choice-node or an entry. We will in this paper not describe in full how the binding scheme is designed nor how suspensions are handled.

and-continuation

An and-continuation represents a sequence of program atoms. The first and-continuation allocated in an and-node holds the representation of the guard and body of the guarded goal. An and-continuation is similar in its structure and use to an "environment" in WAM [11]. It holds a continuation program pointer and a tuple of registers.

choice-node

A choice-node represents a choice-box. It contains a list of and-nodes and a *choice-continuation*. A choice continuation represents a sequence of guarded goals and is similar to a "choice-point" in WAM. It holds a program pointer and a copy of the argument registers.

3.2 The worker

A worker is either active and "positioned" in an and-node in the configuration, or idle. The node in which the worker is positioned is called the current and-node. We will also refer to the current choice-node and the current and-continuation.

To install itself in an and-node a worker must add its identifier to the set of identifiers held by the and-node. The set of identifiers allows a worker to determine if it is alone in an and-box. It is also possible to determine exactly which workers are installed in an and-node, a possibility that can be utilised by the garbage collector. When the worker is de-installed from the and-node the identifier is removed.

The worker is driven by a set of *tasks*, where each task is associated with an and-node in the configuration. The worker must handle all tasks in the current and-node before it is allowed to move up to the parent choice-node. During the

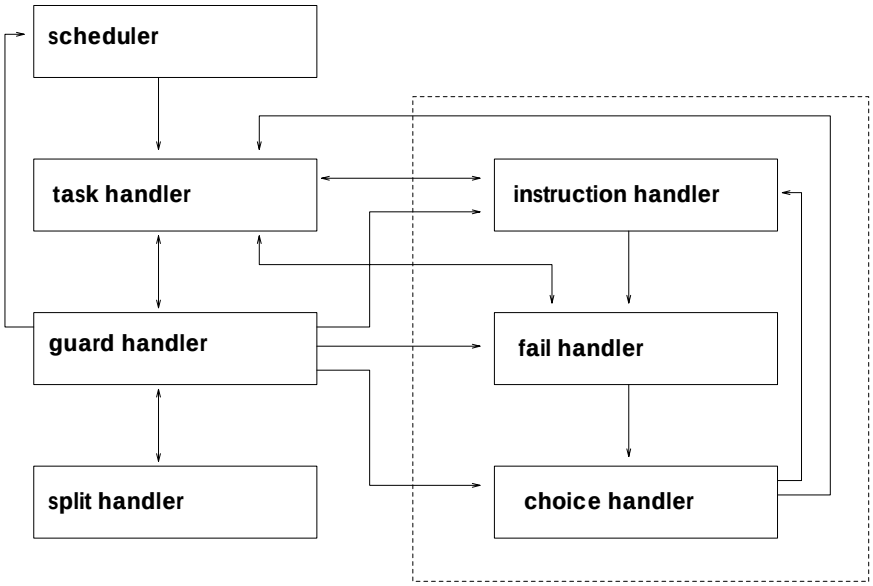

Figure 3: the abstract machine states

execution of a task new tasks can be generate. All new tasks will be associated with nodes in the subtree formed by the current and-node.

By ensuring that a worker always performs all tasks in a box before it can move up, half of the problem of detecting stability have 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.

3.3 The states

During execution a worker moves between different handlers as illustrated in Fig 3. The handlers are: *instruction handler*, *task handler*, *fail handler*, *choice handler*, *guard handler* and the *scheduler*.

The instruction handler is the core of the abstract machine and can together with the fail handler and choice handler be seen as the WAM part of the abstract machine. Notice that the instruction handler is only one part in the abstract machine, a part important to implement as efficiently as possible using compilation techniques as well as implementation techniques. Our work is however focused, not on the instruction handler, but on the implementation of and interaction between the other handlers since these are unique to our system.

3.3.1 instruction handler

The instruction handler is responsible for calls, selection of guarded goals, building of data structures and open code unification. The instructions for data manipulation are, apart from variable bindings, very similar to the instructions in the WAM. We will not in this paper describe all these instructions, but only describe the instructions that are important for the understanding of the execution model.

A definition consist of decision code that is terminated either by: a *suspend* instruction, a *single* instruction, a sequence of *try/retry/trust* instructions, a *fail* instruction or the code of a body. The single instruction and try/retry/trust instructions determine which guarded goals should be executed.

The code of a guarded goal consists of an *allocate* instruction, a sequence of *call* instructions, a *guard* instruction followed by either a sequence of call instructions and a *deallocate-execute* instruction or a *deallocate-proceed* instruction. Below is an outline of the code for a definition where the instructions are shown without their arguments.

```
     :
    jump L0

L0  try G0
     :

G0  allocate
     :
    call
     :
    guard
     :
    call
     :
    deallocate-execute
```

If the decision code can determine that the guard is solved, the body can be *directly promoted*. Only the body of the guarded goal is coded resulting in either an allocate instruction, a sequence of call instructions followed by a deallocate-execute instruction or, if there is only one goal in the body, a single execute instruction or, if there are no goals in the body, a proceed instruction. Below is an outline of a definition with a directly promoted body.

```
     :
    jump L0
```

```
L0 allocate
    :
   call
    :
   deallocate-execute
```

Notice that an allocate instruction always is present before a guard instruction in the code of a guarded goal and always in the code of a body if there are more than one call instruction in the body. This is different from how the allocate instruction is issued in the WAM [11] where an allocate instruction is issued only if "permanent" variables are present.

The instruction decoding phase needs a reference to an *insertion point*. The insertion point is a location between two elements in the sequence of goals, and is necessary in order to keep the sequence ordered. The most important instructions without their arguments:

call/execute/deallocate-execute: A call instruction initiates the call of a program atom. The continuation program pointer of the current and-continuation is updated and the insertion point is set to the left of the current and-continuation. A continuation task is added for the and-continuation and the worker starts decoding the instructions from the definition.

The deallocate-execute instruction is used for the last call in a body. It removes the and-continuation and sets the insertion point to the position of the removed and-continuation. An execute instruction is used for a directly promoted single goal body and does not change the insertion point.

single, try/retry/trust: The single instruction creates a choice-node with a single and-node and inserts it at the insertion point. The and-node is the current and-node and the insertion point is the first position in the (empty) sequence of goals.

The try, retry and trust instructions are used when more than one guarded goal is applicable. The try instruction creates, apart from a choice-node and an and-node, a choice-continuation where it saves the argument registers and a continuation program pointer. The retry instruction will update the continuation program pointer and the trust instruction will remove the choice-continuation.

fail: The fail instruction is used when no applicable guarded goal is found. The worker will enter the fail handler.

suspend: The suspend instruction is used if the decision code can detect an immediate suspension. A choice-node with a choice-continuation is created and inserted at the current insertion point. The argument registers are

saved and the continuation program pointer is set to the first instruction in the decision code. A suspension is added to the variable (only one) on which the goals is suspended and the worker then enters the task handler.

allocate: An and-continuation is created and inserted at the insertion point. Note that the and-continuation can be the first instruction of a guarded goal or the first instruction in a body that has been directly promoted. The and-continuation becomes the current and-continuation.

guard: The guard instruction terminates the instructions that belong to the guard. The continuation program pointer in the current and-continuation is updated and the and-continuation is inserted as the body of the current and-node. The worker then enters the guard handler.

deallocate-proceed/proceed: The deallocate-proceed instruction is used as the last instruction of a body without any goals, where an and-continuation was allocated in the guard. The instruction removes the current and-continuation and the worker enters to the task handler. The proceed instruction is the last instruction in an empty body that has been directly promoted. The worker enters the task-handler.

During unification of an external variable, a suspension for an unifier entry is created and added to the variable. Variables can therefore have suspensions referring to entries or suspended choice-nodes. During unification the suspensions will generates wake and recall task that are added to the set of tasks for the current and-node. If a unification fails the worker will enter the fail handler.

3.3.2 task handler

The tasks are the primitive operators for the concurrent computation. There are three kinds of tasks: continue, recall and wake. Continue tasks are generated by the call instruction, wake and recall tasks are generated when a variable with suspensions is bound. Each task is associated with the and-node in which the task is generated.

A worker will examine the tasks associated with the current and-node and select one of them. If no more tasks exist for the and-node the worker will enter the guard-handler. The different tasks are:

continue: The continue task refers to an and-continuation in the current and-node which is made the current and-continuation. The worker will enter the instruction handler starting with the instruction pointed to by the continuation program pointer of the and-continuation.

recall: A recall task refers to a choice-node that has been created by a suspend instruction. The choice-node is removed and the saved registers are copied to the argument registers. The worker enters instruction decoding starting

with the instruction pointed to by the continuation program pointer. The insertion point is the position of the removed choice-node.

wake: A wake task refers to an entry of an and-node immediately below the current and-node (or rather below a choice-node below the current and-node). The worker must install itself in the and-node, remove and examine the entry. The entry can, as said before, be of four different kinds: unifier, suspension, continuation or dead.

A unifier entry is retried by unifying the (now bound) variable and the term. If the unification fails, the worker moves to the fail handler, if it succeeds the entry is marked as dead and the worker will continue in the task handler.

For both suspension entries and continuation entries, the set of suspensions are transformed to tasks and the entries are marked as dead. A dead entry can be ignored since it has already has been taken care of. Execution will proceeds in the task handler.

3.3.3 guard handler

The worker enters the guard handler after all tasks associated with the and-node are processed. When the worker first enters the guard handler it locks the parent choice-node and current and-node (in this order). The worker then determines if it is alone in the current and-node. If it is not alone it will, if the current and-node is the main and-node, move to the scheduler state. If it is not the main and-node the worker will de-install itself from the and-node, release the lock of the node and move to the choice handler. Notice that the lock of the choice-node is still held.

If the worker is alone in the and-node it will examine the type of the and-node. The type can be either: failed, pruned, main or a guard operator.

failed: A failed type means that another worker has failed the and-box but has left to the last worker to clean up. The worker moves to the fail handler.

pruned: A pruned type means that another worker has pruned the and-node, the worker will move up to the parent choice-node and enter the choice handler.

main: If this and-node is reached, there is nothing more to do and the worker enters the scheduler state.

guard operator: If the type is a regular guard operator the worker tries to perform a pruning or promotion operation. In a pruning operation the choice-continuation of the parent choice-node is removed. Any pruned sibling and-nodes are marked as pruned. A promotion operation replaces the parent choice-node with the body of the current and-node, the worker

then enters the instruction handler and starts with the first instruction in the and-continuation.

If promotion is not allowed the worker must examine if a split operation can be performed. If the and-node is stable and a candidate is found the worker moves to the split handler. If a split operation is not possible, the worker is done. It moves to the parent choice-node and enters the choice-handler.

3.3.4 choice handler

In the choice handler the worker looks for a choice-continuation. If a choice-continuation exists, the saved registers are copied back to the argument registers. The worker then enters the instruction decoder starting with the instruction pointed to by the continuation program pointer. If no choice-continuation exists, the worker releases the lock of the choice-node, moves to the parent and-node and enters the task-handler.

3.3.5 fail handler

The fail handler is entered from the instruction handler, guard handler or task handler. If the worker enters from the guard handler the locks are already taken, if it enters from the instruction handler or task hander it must first take the locks of the parent choice-node and current and-node (in this order). Once the lock of the current and-node is taken the workers determines if it is alone in the and-node.

If the worker is not alone in it examines the type of the and-node. If the type is either failed or pruned, the worker simply de-installs itself from the and-node, releases its lock, moves to the parent choice-node and enters the choice-handler. If the type is a proper guard operator, the node is first marked as failed. If the node is the main and-node, the worker will de-install itself, release the lock of both the and-node and the choice-node and enter the scheduler.

If the worker is alone in the and-node and the and-node is pruned, the worker de-installs itself, moves to the parent and-node and enters the task-handler. If the and-node is already failed or the type is a regular guard operator the worker will mark the and-node as failed, remove it form the configuration.

If the worker has removed the node it will check if the node represented the last and-node of the parent choice-node. If this is the case the worker will remove the parent choice node, enter the parent and-node and again invoke the fail handler. If only one sibling and-node exist this and-node could be promoted. The worker will therefore adopt the node as the current and-node and proceed in the task handler. If none of the special cases apply the worker will move up to the choice-node and enter the choice handler.

3.3.6 split handler

When the worker enters the split handler, it holds the lock of the current and-node and parent choice node. The worker will first position itself in the parent and-node of the candidate and-node, called the mother-node (this might mean that the worker will have to move down in the tree but the mother-node often turns out to be identical with the current and-node). It then ensures that the mother node and its parent choice node are locked and then inserts a new locked and-node, called the copy-node, to the left of the mother-node. Once the copy-node is inserted, the lock of the parent choice-node can be released to allow other workers to make split operations in sibling nodes.

After having copied the contents of the mother-node into the copy-node, the worker will release the locks of the mother node, enter the copy of the candidate, release the lock of the copy node and move to the guard handler. Before this is done, it will however add continuation entries and tasks to make sure that no deterministic computations are forgotten in the mother-node. If the candidate had only one sibling and-node this node is deterministic and can, if it is solved, be promoted. To find this task, an empty continuation entry is added to the sibling node referenced by a continuation entry in the mother node. If the candidate had more than one sibling but the mother-node is stable an empty continuation entry is added to the mother-node. If an entry is added to the mother-node a wake, task for the entry is added to the set of tasks of the mother-nodes parent and-node.

3.3.7 scheduler

The worker in this state has nothing to do. To perform parallel computation, the scheduler will supply another task to do from another worker. Once a task is found and assigned to a worker, the worker enters the task handler. If all workers are in scheduler state, the whole process will stop.

More detail will be given in section 5.

4 Implementation

In this section we will describe some of the implementation specific details of the representation of nodes in the execution state. We will describe a scheme that has few locking operations and allows nodes to be reclaimed explicitly.

We will use an atomic swap operation on a single word to implement locking primitives. The locks will not hold simple locked/unlocked values but rather hold values as for example locked/dead/pointer. All locks will be spin-locks, i.e., a worker will swap the value of the lock until the lock is taken. This does of course not guarantee progress of each worker but is not a limitation in practise.

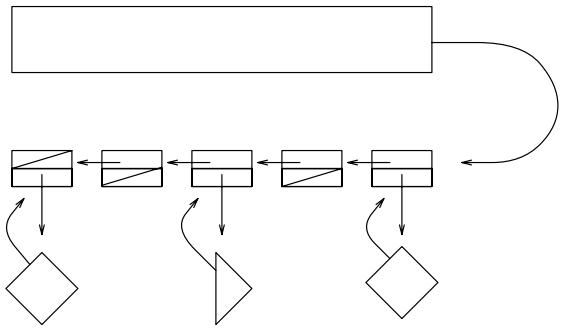

Figure 4: the representation of the sequence of goals

4.1 goals

The sequence of goals in the guard is represented by a single linked list of *insertion cells*. Each insertion cell is tagged and is either *dead*, pointing to a *choice-node* or pointing to an *and-continuation*. The choice-nodes and and-continuations also hold a pointer to the insertion cell. 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. Figure 4 shows the structure of a list of goals.

When a call instruction is executed a new cell is inserted after the insertion cell of the current and-continuation. The new cell will be the new insertion point. No locking is necessary since the worker is the only one that has access to the and-continuation and hence is the only one that will insert a new cell.

When a body is promoted the and-continuation will inherit the insertion cell of the parent choice-node. The last instruction of an and-continuation will mark the insertion point as dead (deallocate-proceed) or reuse it for the last goal (deallocate-execute instruction).

The advantage of the scheme is that no locking is necessary in order to keep the list ordered. A disadvantage is that the list of insertion cells will grow and 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 only by the last worker to leave the and-node. Dead entries can be removed at any point in the execution, during execution or by garbage collection.

4.2 guards

The operations on the list of and-nodes is different from the operations on the list of goals. A worker must be able to remove (by pruning or failure) any and-node and this will require some locking. A fast insertion operation, provided in the list of goals, is not essential since new and-nodes are only created from

a single choice-continuation and in the split operation and both operations are infrequent.

The and-nodes will be linked in the list starting with the rightmost and-node. The lock in the choice-node will control all modifications on the list of and-nodes and also of the choice-continuation. The list is linked from right to left in order to facilitate the insertion of a new and-node in the retry and trust instructions.

Although the insertion and removal of nodes in the list is not crucial, we still want to be able to explicitly reclaim a removed and-node. In order to do this we must protect it from direct references from living data structures. An and-node is referenced (apart from the list of nodes) from variables that are local to the and-node and entries of the and-node. The entries of the and-node will not survive if the and-node is removed but variables will survive if the and-node is promoted. The variables of a promoted and-node is of course not interested in the and-node itself but need the forward pointer to determine their new home.

To solve this problem we extract the forward pointer from the and-node in a structure by its own. This structure will be referred to as the *environment identifier* (or envid for short) of the and-node. The envid contains a three value lock and a forward pointer. The lock can take the values: and-node pointer, locked or dead. The and-node pointer is a pointer to the and-node to which the envid belongs. The forward field is either null or, if the and-node has been promoted, a pointer to the envid of the parent and-node. It is possible to code both the lock and the forward pointer in one word by tagging the different pointers.

When a worker installs itself in an and-node it must first take the lock in the envid. It can then add its identifier to the set of identifiers and release the lock. Only when the worker is installed in the and-node, it is allowed to trust the available information.

removing a node

If a worker wants to remove a failed and-node, it has already taken the lock of the parent choice-node and the lock of the and-node. The lock on the and-node prevents any other worker from entering the node, the lock of the choice-node prevents any other worker from modifying the list of and-nodes, i.e., prune the node. The and-node can thus be removed from the list and reclaimed.

Similarly, if a worker wants to prune sibling and-nodes it has already taken the locks of the parent choice-node and current and-node. It will prune an and-node by first taking the lock of the node, mark it as pruned and if no other worker is present, remove and reclaim the node. Another worker might hold the lock of the and-node to be pruned but it is then only held for entering or leaving the node. Any worker that wants to fail or prune the and-node is waiting for the lock of the parent choice-node, the locking scheme is thus dead-lock free.

In a promotion operation, the current and-node and its parent choice node

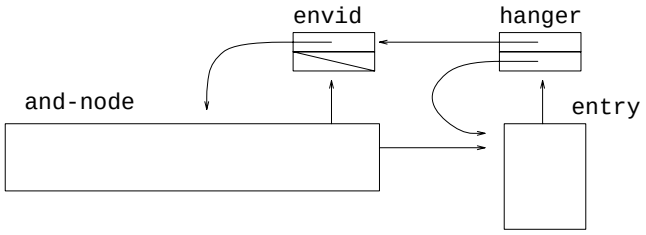

Figure 5: the representation of an and-node

can safely be reclaimed, no other worker will access the choice-node nor the and-node before having taken the lock of the and-node.

sub-trees and entries

When an and-node is reclaimed the lock of the envid is set to dead. Other workers that are waiting for the lock will detect the dead lock and proceed with other tasks. The worker can then reclaim not only the and-node but also all nodes below it in the execution state. The hierarchical structure of the suspensions guarantees that no other worker will try to enter the sub-tree so these operations can be done without any locking.

Entries of an and-node are also structures that we want to reclaim explicitly. We protect each entry with a *hanger*, which contains a pointer to the envid of the and-node and a pointer to the entry. Figure 5 shows the representation of an and-node with single entry. When a worker examines a wake task it will first try to lock the envid pointer of the hanger. If the lock is already taken it will ignore the entry. If the lock is seized the worker will try to take the lock of the envid, install itself in the and-node, remove the entry from the list and unlock the and-node.

4.3 reclaiming structures

We have described a scheme that allows a worker to explicitly reclaim nodes in the execution state. By reclaiming nodes explicitly we will decrease the garbage collection time and more importantly improve cache performance.

Choice-nodes, and-nodes and entries have different size but they all have a fixed size and can thus easily be maintained in free-lists. And-continuations and choice-continuations can have any size and if these are to be reclaimed we have different choices. One possibility is to break the continuations up into fixed size segments. This has the drawback that it is harder to access the continuations since the registers are not in consecutive order.

Access by instructions to the and-continuations is most important to implement efficiently. Since all accesses by instructions are known at compile time the

compiler can be made aware of the segmented representation and issue special instructions for the access that are not in the first segment. If these accesses are infrequent the only drawback is that the number of instructions is increased. This can be avoided by adding only two new instructions: one that will set the and-continuation register pointer to the right segment and another that will reset it. This will of course slow down the access to registers not in the first segment even more, but it could pay off depending on the technique used in the implementation of the instruction handler.

the size of a block

If all structures could use the same size blocks much would be gained. It is of course easier to maintain one free-list but it will also improve cache performance. For example, if the memory used by a reclaimed and-node immediately can be used for a constructed choice-node, this memory segment will most certainly be in the cache.

We will of course lose some space if we use the size of the largest structure to represent even the smallest structure but this is not so important. The important thing is to avoid *false sharing*. False sharing occurs in a shared memory multiprocessor when two processors access different data structures that happens to be allocated in the same cache-line.

The size of a cache-line of the second level cache of the SPARCcenter-2000 (our target machine), is 64 bytes. Choice-nodes, and-nodes and entries all fit into 64 bytes. If the continuations are broken up into 64 byte segments each segment will hold a maximum of 15 register. The first segments will hold even less (choice-continuation 13, and-continuation 10) since these segments also hold other pointers. We have however found that this is not a serious limitation.

5 Scheduling

The scheduler is invoked when a worker enters the scheduler state. The scheduler serves two purposes: dynamic load balancing and the detection of termination.

5.1 Dynamic load balancing

As we mentioned in subsection 3.3.2, the tasks are the primitive operations of concurrent computation and can be computed in parallel. To balance the load of the workers, the scheduler supply one or more tasks to an idle worker which is in scheduler state. The sources of the tasks are the *local task stack* and the *global task queue*.

local task stack

Normally, a task generated by a worker is pushed on the local task stack owned by the worker. This operation does not need any locking operations

since the stack is private to the worker. The task handler will pop tasks from the local task stack.

global task queue

The global task queue is utilized to save the tasks which are not owned by any workers, for example, tasks generated by the operating system. The global queue can also be used as a pool of tasks that could be given to idle workers.

The use of the local task stack is similar to the use of task stacks in the sequential implementation of AKL [7].

To enable parallel computation, tasks must be moved between stacks from busy workers to idle workers, directly or via the global queue. A busy worker should be able to continue its work with as little interference as possible. If no worker is idle, there should be no overhead caused by the scheduler. On the other hand we would like a idle worker to get a new task as soon as possible. The new task should be found without disturbing any busy worker. We have several questions to answer when the scheduler is designed:

- Should a busy worker be interrupted, or should it voluntarily move tasks to the global queue?
- If we allow an idle worker to interrupt a busy worker, should the idle worker steal tasks from the busy worker's stack or should the busy worker move the tasks to the global queue?
- How many tasks should be moved from a busy worker and/or how many tasks should the worker keep?
- Should we look for a scheduler that can often give the best performance or are we happy with average but predictable performance?

To construct a perfect scheduler is difficult because there are several parameters to be tuned and the result obtained depend on the properties of the executed program. Our experience is however that even a simple scheduler works well in most cases, only in extreme cases must more advanced schedulers be used. We have experimented with three different schedulers:

steal An idle worker will steal one task from any busy worker. This requires that the local stack is protected by a lock which limits the workers freedom to pop a task from its own task stack.

This scheduler is very eager and typical parallel benchmark problems are solved in very short time. The problem is if too many workers are idle and few tasks are available. The system will then show very bad performance due to thrashing.

voluntary Busy workers periodically move tasks to the global task queue. The idle workers just take tasks from the global queue.

This scheduler has good performance for programs with fine grained concurrency. Workers will have a small overhead for moving task to the global queue but will spend very little time in the scheduler.

signal An idle worker will send a signal to all busy workers and wait for a task to be added to the global queue. The busy workers will have to monitor this signal but this can be coded in existing signal handling and does therefore not cause any additional overhead.

The scheduler shows good predictable performance, the drawback is that an idle worker must wait for some time in the scheduler for a task to be added.

Notice that there is no difference between AND and OR parallel tasks. They are all handled in a uniform scheduler.

5.2 Detection of Termination

The detection of termination is difficult problem in parallel computation. In the current implementation, the scheduler uses a *busy worker bit field* to detect termination. Each worker clears its bit when it enters the scheduler. A worker in the scheduler can detect termination by checking if all bits are zero.

In addition to the busy worker bit field the scheduler checks operating system related tasks, e.g., if a suspended goal is waiting for a blocked network stream or user terminal. The operating system can generate tasks in the future that must be handled, the execution must therefore suspend until no such tasks can be added.

6 Evaluation

This section is not a complete evaluation of the system, it is only intended as an overview of how the Penny system compares to other systems and how it performs on a multi processor architecture. All benchmarks were executed on a SPARCcenter-2000 running SunOs 5.4.

6.1 KLIC vs. Penny

To evaluate how implicit parallelism compares to explicit parallelism the Penny system was compared to the KLIC system [8]. The KLIC compiler compiles KL1 programs into C and produces very fast code. A fine grained concurrent benchmark was chosen since this will stress the scheduler of the Penny system.

Table 6 shows the performance of the KL1 "life" benchmark. The timings were the best in ten runs. For the KLIC system the ten runs were all done in

Proc	1	2	3	4	6
KLIC	1169	734	663	387	357
Penny	1775	1293	954	809	660
P:K ratio	1.5	1.8	1.4	2.1	1.8

Figure 6: The game of life (time in milliseconds)

one execution to avoid system time overhead in the initialization of heap areas. In the Penny system heap allocation is done at start up and is not included in the reported execution time.

The KL1 version of "life" is a reduced game of life where each cell only has four neighbors. The grid is divided into clusters and the clusters are then distributed on the available processors. We ran the experiment with a 30x40 grid divided into twelve clusters of 10x10 cells each. The division allows an even distribution of clusters on available processors.

In the Penny version no annotations are necessary to parallelize the program. The program is simplified since there is no need to divide the program up into clusters, all cells are treated equally. A benchmark like the life program could be the death of an implicit parallel system like Penny but it turns out that the Penny system performs extremely well considering the difference in implementation technology.

The figures in the table 6 show that the Penny system, for this benchmark, only is twice slower than the KLIC system while the normal factor between the KLIC system and Penny is around four to six (sometimes up to ten). There are several possible reasons for the life benchmark shows so good results for the Penny system.

- The KLIC system is much faster on instruction decoding since it does not have the overhead of an emulator. Since a large part of the execution time is spent in other parts of the system the instruction decoding handler is not so important for this benchmark.

- The KLIC system allocates suspended goals on the heap and does not explicitly reclaim the structures once they are re-executed. The cache performance of the Penny system could be better in this respect but these effects are very hard to measure.

- Binding shared variables (shared between nodes) is in the KLIC system considerably more expensive than binding non-shared variables. In the life benchmark about 10% of the communication (through variables) is performed with shared variables. However, experiments with different cluster size indicate that this the KLIC system is fairly robust in its performance,

Proc	1	2	4	6
MUSE	2.8	1.5	1.0	0.7
Penny	21	11	5.8	4.0
P:M ratio	7.5	7.3	5.8	5.7

Figure 7: Nine queens (time in seconds)

only when the grid is divided into very small clusters does a decrease in performance occur.

The main deficiency of the Penny system compared to the KLIC system is its naive way of handling builtin procedure such as arithmetic. The Penny system must also handle the simple guards as deep computations since the compiler will not detect the possibility to make a "flat suspension".

6.2 MUSE vs. Penny

To evaluate how the Penny system performs for non-deterministic computations the system was compared to MUSE, an or-parallel Prolog system available with SICStus 3.0 [10].

Table 7 shows the performance of a "nine queens" benchmark. As is clearly seen the MUSE system outperforms the Penny system by almost a factor eight. This is not surprising since the MUSE system uses back-tracking whereas the Penny system uses copying to perform non-deterministic computations. Notice however that the relative performance decreases, an indication that the Penny system scales up well when the number of processors are increased.

In this version of the queens program no intelligent constraint handling is performed nor is concurrency used, so the Penny system will not make up for the copying overhead with a smaller search space or increased parallelism. This benchmark was selected because it is a worst case comparison (worst for Penny) of the two systems.

By using a benchmark where the Penny system could benefit from its more advanced search strategies, Penny would be put in a better light but this would be to compare apples and pears.

The MUSE system is, like the Penny system, an emulated system. The native-code SICStus compiler will run the same benchmark in 1.5 seconds on the same machine.

6.3 Scaling

To show how well the system scales when the number of processors are increased a set of benchmarks were executed on a SPARCcenter-2000 with twenty proces-

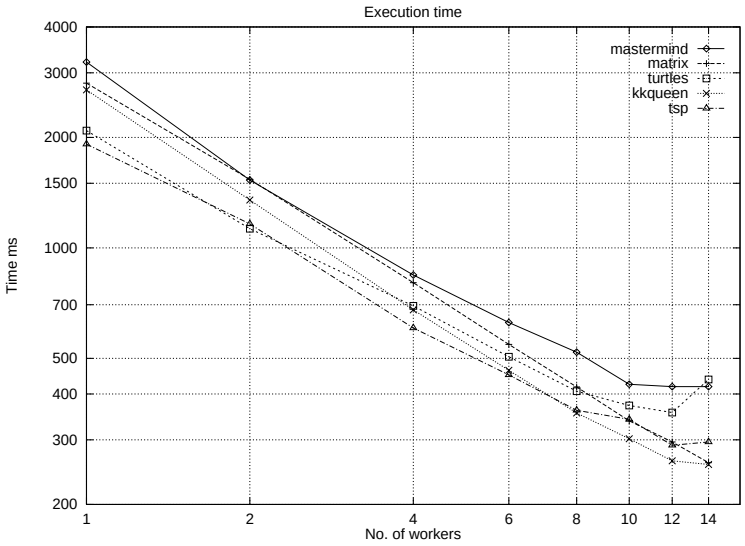

Figure 8: deterministic benchmarks

sors. We did not have exclusive control of the machine so the benchmarks were executed while other users where using the machine (four processors where used for a ball-bearing simulation :-).

Figure 8 shows five deterministic benchmarks often used in the committed choice community. A log-log scale is used to clearly show the decrese in performance fwhen more than twelve processos are used.

mastermind: The mastermind puzzle by E Tick, using two guesses and three colours.

matrix: A 500x500 matrix (filled with ones to avoid using big-nums) multiplied by a vector.

turtles: The turtles puzzle by E. Tick.

kkqueen: The candidates/non-candidates queens program by K. Kumon/E. Tick, using 9 queens.

tsp: A near-optimal solution to the traveling salesman problem by M. Veanes, using 24 nodes.

The figure shows that the system scales up rather well even for these small benchmarks. Problems only start to occur when twelve processors are used. The benchmarks where executed using the stealing scheduler. We expect the

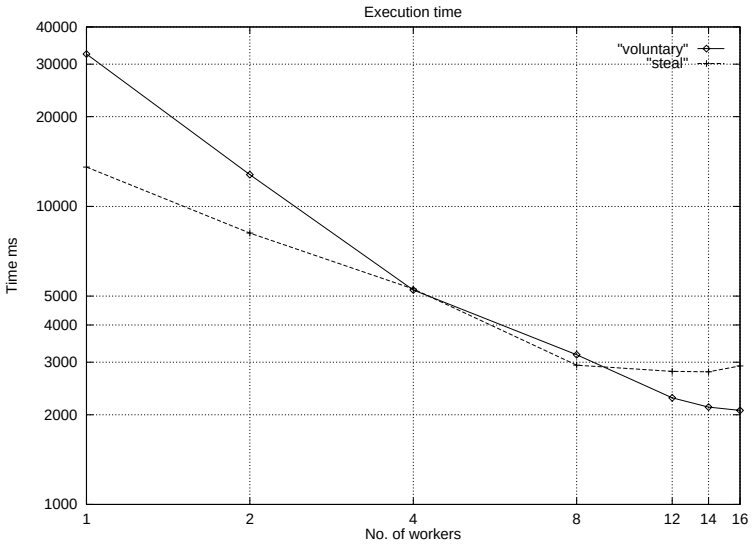

Figure 9: Scheduler comparison

voluntary scheduler to perform better but we have no results available for these benchmarks at this point.

The benchmarks can all be executed from left to right without any suspensions and can therefore be executed as Prolog programs. The execution time for the benchmarks using the emulated-code compiler in SICStus 3.0 [10] is two to three times smaller. This is mainly due to better compilation techniques and far better handling of builtin procedures.

6.4 Scheduling

We have only preliminary results for the different schedulers. Figure 9 shows a game of life benchmarks (all eight neighbors) using the steal and voluntary schedulers. As is clearly seen the voluntary scheduler has very big problems when one or two processors are used. This is because the worker spends most of its time moving task to and from the global task queue. When more processors are used, the voluntary sharing of work pays off and when more than eight processors are used the voluntary scheduler outperforms the steal scheduler.

These figures are very preliminary, we know how to avoid the initial overhead for the voluntary scheduler but have not yet been able to redo the benchmark.

7 Summary

We have described the fundamental components of a parallel implementation of AKL that allows both and- and or-parallel execution. The main advantageous features of the implementation are the representation of the execution state to avoid locks as much as possible, the memory management method to reclaim almost all administrative structures, and its flexible and powerful scheduling scheme.

We also presented promising performance results comparing the system to KLIC and MUSE, which are two of the fastest parallel logical language system, as well as the preliminary scheduler performance.

further investigation

To compete with systems such as KLIC and SICStus the Penny system needs a better compiler. A compiler that can generate better decission code and handle builtin procedures efficiently will raise the performance to the level of emulated SICStus or half the speed of KLIC. This will hopefully be acheived when the compiler developed by Per Brand [3] is integrated in the system.

The system can today only handle unification, it is an open question if the binding scheme can be adapted to support, for example, a finite domain constraint system. This will hopefully be investigated in the near future.

acknowledgments and remarks

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

Some of the evaluations have been done on the SPARCcenter-2000 of IDA at the University of Linköping. A special thanks to Prof. Peter Fritzson who has given us the opportunity to use the machine.

References

[1] Khayri A. M. Ali. A parallel copying garbage collection scheme for shared-memory multiprocessors. *New Generation Computing*, 13(4), December 1995.

[2] Galal Atlam and Johan Montelius. Parallelization of Garbage Collection in a CCP System, Penny. Acclaim deliverable 4.1/3, SICS, June 1995.

[3] Per Brand. A Decision Graph Algorithm for CCP Languages. Acclaim deliverable 4.3/2, SICS, June 1995.

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

[5] Sverker Janson. AKL A Multiparadigm Programming Language. Uppsala Thesis in Computing Science 19, SICS Dissertaion Series 14, Uppsala University, SICS, 1994.

[6] 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.

[7] 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.

[8] KLIC. klic-requests@icot.or.jp, ICOT.

[9] Johan Montelius and Khayri A. M. Ali. An and/or-parallel implemetation of akl. *New Generation Computing*, 13(4), December 1995.

[10] SICStus v3. URL http://www.sics.se/ps/sicstus.html, SICS.

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