tion. The benchmarks are the familiar naive reverse, which is called with 300 and 1000 elements, and merge sort by O'Keefe (system sort/2 in SICStus and in AGENTS), which is called with lists of integers of length 1000 (medi) and 11240 (maxi), formed by the decimals of π in groups of five.

The remaining difference in speed between AGENTS and SICStus seems to be mainly due to the more optimised emulator of SICStus. In the case of sort, register allocation in AGENTS is inferior and there are flat guards that cannot be optimised by the simple scheme presented. In both cases, the better ratio for the optimised case with larger inputs is probably due to the use of a copying garbage collector in AGENTS. For the small inputs, no garbage collection is needed.

The optimisation discussed deals only with a very simple case. A scheme for clause selection and quietness detection that optimises all flat guards is given by the decision graph method for arbitrary constraints by Brand [1994].

6.7.2 Small Variables

An unbound variable without suspensions needs only the home and state attributes. We call this a *small variable* (as opposed to *large* variables).

In an implementation with tagged pointers, a small variable can be stored in place of a reference. In particular, small variables can be used as arguments to constructors and in y-registers in and-continuations.

When binding a small variable, it is replaced by a reference to the value. If the variable is external, the home of the variable is still needed. There are two alternatives: (1) replace the small variable with a reference to a new large variable before binding, and (2) trail also the home of the variable. Both are feasible. In the latter case, the home and value attributes can share the same memory location also in the large variables.

If small variables are used as arguments to constructors, a tree built incrementally, by creating incomplete nodes with variable arguments that are later bound, will not contain any superfluous nodes. When they are not, e.g., in a list produced by append, there is one variable “between” every list cell. But, when they are used, care must be taken to make a reference to the argument if a small variable is encountered when accessing arguments.

If small variables are used in y-registers, the variables created for “return values” of calls are stored there. A non-suspending computation employing the flat guard optimisation will only use heap storage for building constructors. But, care must be taken to *globalise* these to avoid dangling references (cf., local variables and unsafe values in the WAM).

AGENTS has small variables, but does not (yet) use them in y-registers.

6.7.3 Separate Read and Write Code Streams

The instructions for tree constraints are suboptimal. Schemes based on separate code streams for reading and writing [Mariën and Demoen 1991] are more ef-

ficient and make it possible always to bind variables to complete tree constructors, instead of partially built tree constructors as in the get_constructor and put_constructor instructions. The latter feature solves two problems with the code described.

AGENTS provides *generic variables*, which are used to parameterise the system with different constraint systems. These are equipped with a method table, and when bound call a unify method. In principle, it is possible to generalise the constraint system of trees, e.g., to records [Smolka and Treinen 1992] or to RT [Keisu 1994], by introducing generic variables for these domains that "understand" what is means to be unified with the special case of a tree. However, for this it is necessary that variables are bound to complete tree constructors.

A parallel version of the abstract machine has to take care when binding variables. A common technique (e.g., [Crammond 1990]) is to atomic-swap the state and value attributes of the variable with the new value, and, if it was discovered that the variable was already bound, to unify the old and new values. For this it is also necessary to bind to complete tree constructors.

6.7.4 Trimming and Re-using And-Continuations

Non-tail-recursive definitions grow the and-stack. As described, the call instructions do not perform "trimming" of and-continuations, as of environments in the WAM. This would be easy to introduce, in particular since trimming can take place unconditionally. It is possible both to shrink and grow and-continuations as needed.

Tail-recursive definitions are in every sense like loops, but the and-continuations are pushed and popped in every cycle. This is shared with the WAM. Meier [1991] and others have suggested that environments (corresponding to and-continuations) should be reused, in the way a stack frame normally survives the entire procedure invocation in a procedural language. For this, Prolog requires determinacy analysis, since and-continuations are involved in nondeterminism, and may have to be shared between solutions to goals in the "loop". In AKL, and-continuations are never shared (not even in the sharing scheme discussed in the following section) and a corresponding scheme can be designed and implemented with less difficulty.

6.8 COPYING

The WAM shares parts of the goal clause between the branches in a search tree, using the trail, the choice-point stack, and backtracking. The abstract machine described here solves this by copying. Naturally, nondeterminism thus provided cannot be used like that of Prolog.

6.8.1 A Truly Bad Case

Observe, in Figure 6.7, the cost in AGENTS of enumerating all tails of a list using the following definition. This is intentionally a very bad case.

```
tail(X, X).
tail(X, [_ | Y]) :- tail(X, Y).
```

The testing environment is as for the benchmarks in Section 6.7.1. The times presented (in milliseconds) are the average of three consecutive runs of

```
bagof(X, tail(X, L), S)
```

A reference to S is live during execution. The heap is not reset between runs. The total is divided into times for copying, garbage collection, and other.

Length	Total	Copy	% Copy	GC	% GC	Other	% Other
300	309	237	77	38	12	34	11.1
600	1098	832	76	208	18	64	5.8
900	2518	1886	75	553	22	74	2.9
1200	4455	3441	77	878	20	137	3.1
1500	7268	5537	76	1560	21	171	2.4
1800	12293	8438	69	3583	29	272	2.2

Figure 6.7. AGENTS: Cost of enumerating all tails for different length lists

Length	Failure	Findall	Findall2	GC	% GC
300	3.3	334	333	0	0
600	6.0	1307	1283	0	0
900	9.6	2877	4058	1154	28
1200	12.9	5048	8247	3129	38
1500	15.6	7927	19769	11833	60
1800	18.6	11475	20946	9250	44

Figure 6.8. SICStus: Cost of enumerating all tails for different length lists

We now compare this with SICStus Prolog (Figure 6.8). The times (in milliseconds) are the averages of three runs. “Failure” is the total execution time (including GC) for a failure-driven loop of the form

```
tail(_, L), fail
```

“Findall” is the runtime for a goal of the form

```
findall(X, tail(X, L), S)
```

The heap is reset between runs, and there is no garbage collection. “Findall2” is the time when the heap is not reset, and garbage collection becomes visible.

AGENTS is, for this truly bad case, between two and three orders of magnitude slower than SICStus with plain failure-driven enumeration, and grows quadratically in the length of the list. The cost of activities other than copying and garbage collection grows more or less linearly with the length of the list, as could be expected.

In AGENTS, the elements are enumerated using bagof. Since copying has already been done, this carries no extra cost, whereas SICStus exhibits a dramatic, but non-surprising, slowdown when going from plain failure-driven enumeration to findall. (SICStus bagof is almost twice as slow.)

6.8.2 Some Better Cases

For problem solving purposes, the copying cost seems to be manageable, and stays around 40–65% of the total execution time (as illustrated by Figure 6.9).

The examples are: queens(N) is the solution for the N-queens problem presented in Section 3.5, using a centre out heuristic. money(dif) is the SEND+MORE=MONEY cryptarithmetic puzzle. zebra(cir) is the Five Houses puzzle. substitution and frequency are solutions to a substitution cipher from Yang [1989]. knights(6) and allknights(5) find one and all solutions to the knight's tour problem on boards of size 6 and 5, respectively. The programs are rather naïve in that they use tree constraints only. Solutions based on AGENTS with FD-constraints are quite competitive with CLP systems in speed, and have a similar copying overhead [Carlson, Haridi, and Janson 1994].) The columns are as before, with a new column for the number of copying steps performed.

Example	Total	Copy	%Cp	GC	%GC	Other	%Oth	Cp:s
queens(4)	16	4	25	0	0	12	75	1
queens(8)	89	31	35	7	8	51	57	4
queens(16)	608	286	47	120	20	202	33	8
money(dif)	260	172	66	9	3	79	31	152
zebra(cir)	88	55	62	5	6	28	32	29
substitution	1758	1186	67	263	15	309	18	182
frequency	229	127	55	47	21	54	24	45
knights(5)	131	38	29	6	5	87	66	24
knights(6)	973	518	53	80	8	375	39	201
allknights(5)	15870	4015	25	997	6	10858	69	3213

Figure 6.9. Copying cost for some problem solving programs

The reason for the reasonable behaviour of problem solving programs is that for most guessing steps most or all of the copy is actually used. This is particu-

larly true for all-solutions search. In one-solution search, if the solution is found early, more copying will be wasted.

6.8.3 A General Sharing Scheme

A remedy for the copying overhead could be a more general sharing scheme for machine states of the type described here. Such ingenious simplicity as that of the sharing scheme of the WAM cannot be hoped for. Already in the transition from plain Prolog execution to execution based on the Basic Andorra Model, as in Andorra-I [Santos Costa, Warren, Yang 1991a], a sharing scheme has to consider updates of objects other than variables. A sharing scheme for AKL, as discussed below, has to deal with a hierarchy in which a new sharing point can be introduced both "above" and "below" existing sharing points.

A complete scheme has not been designed, but the principles discussed here will probably apply. For simplicity, we assume a transition system over trees built of and- and or-nodes. Some transitions copy subtrees. For example,

$$\text{and}(A, \text{or}(B, C)) \Rightarrow \text{or}(\text{and}(A, B), \text{and}(A, C))$$

copies the subtree A. This is similar to the choice splitting rule of AKL. The idea of sharing is that the representation of A will be shared between the newly formed branches.

The first problem is how to make an update of a shared subtree specific to some part of the tree.

The differences between different occurrences of shared subtrees can be maintained in *diff trails*, which are associated with, but are not parts of, and-nodes.

A *diff* has the attributes

- *updated*: reference to a location
- *old*: old contents of location
- *new*: new contents of location

When entering an and-node, the diffs are *installed*, storing the new value in all updated locations. When exiting, the diffs are *deinstalled*, restoring the old value.

When an object that is shared with respect to some ancestor or-node is updated, a corresponding diff is associated with an ancestor and-node of the object that is in the or-node. Which and-node to choose is not obvious. Two possibilities are: (1) the and-node closest to the object and (2) the and-node closest to the or-node. In the first, installation is delayed until working close to the object. In the second, installation is performed once and for all while working in this branch. It is also conceivable to mix these with delayed deinstallation of diffs, where a diff could be installed upon entry of (1), but deinstalled upon exit of (2). For locations that are multiply updated, the scheme can be optimised to avoid multiple diff trailing, e.g., by marking the updated location with a time-stamp for the update. Observe that the diffs trails themselves are subject to sharing.

The next problem is how to distinguish shared subtrees from non-shared subtrees. An approximate scheme is described.

All objects in a tree that are subject to sharing are given *time-stamps*. A time-stamp is a number. Upon creation, or-nodes are given an *age of shared data*, which is equal to the greatest time-stamp in the part to become shared. The *max age* inside an or-node is the maximum age of shared data of it and all ancestor or-nodes. A new object is given a time-stamp that is greater than the max age where it is placed. Thus, any object which has a greater time-stamp than max age is not shared, and can be updated without diff trailing. Other objects may or may not be shared, and should be trailed for safety.

The time-stamp of an object may be represented naively using a slot in the object. A more involved scheme could use the actual address of the object. In both cases, a chunk of objects may be assigned a time-stamp as a group.

Clearly, a scheme of this kind carries a considerable cost. It is not unlikely that it would add an overhead on the execution times of many programs without don't know nondeterminism of, say, 25-50%, but as indicated by Figures 6.7, 6.8, and 6.9, this might pay off for nondeterministic programs, and for an AKL implementation mainly intended for such tasks, a sharing scheme should be considered. For AGENTS, the cost of copying is deemed acceptable, as don't know nondeterminism is mainly intended for constraint programming.

Length	Total	Copy	% Copy	GC	% GC	Other	% Other
300	59	34	57	0.0	0.0	26	43
600	117	65	55	3.2	2.7	49	48
900	166	90	54	6.2	3.7	70	42
1200	228	123	54	7.6	3.3	98	43
1500	290	148	51	16.2	5.6	126	43
1800	366	188	52	35.8	9.8	141	39

Figure 6.10. AGENTS: Cost of enumerating all tails with simple sharing

6.8.4 A Simple Sharing Scheme for Trees

Simply by adding a home attribute to tree constructors, and not copying trees which have constructors external to the copied and-node as roots since it is known that they cannot contain local variables, the truly bad case can be considerably improved. This scheme is available as an option in AGENTS, and the new times are shown in Figure 6.10. The testing conditions are exactly the same as in Section 6.8.1.

Execution time is linear in the length of input data. The execution time is more than one order of magnitude faster than that of SICStus for the same task.

6.8.5 Avoid Copying

An alternative to sharing is to avoid the need for copying. Potentially useful techniques include better selection of candidates, consistency checking, and constraint lifting (also known as constructive disjunction) [Abreu, Pereira, and Codognet 1992; Van Hentenryck, Saraswat, and Deville 1992; Moolenaar and Demoen 1994].

When choice splitting is attempted, the representation of potential candidates is quite explicit. They contain a list of constraints, and the constraints contain variables on which there are suspensions. A candidate can be given a weight according to how many variables it constrains and how many suspensions there are on these, factors that are likely to be correlated with a greater chance of failure of alternatives in sibling choice-nodes, and thereby less nondeterminism.

It is also possible to examine alternatives in sibling choice-nodes before copying, to see if there is some sibling that would fail for the constraints of the current candidate, in which case the candidate can fail without copying. Pertinent siblings can be found via the suspensions.

Finally, all alternatives in a nondeterminate choice-node can be examined, and if they share (i.e., all entail) a constraint, and this constraint is known to cause waking, it can be *lifted* to the parent and-node, instead of copying.

6.9 POSSIBLE VARIATIONS

A few, somewhat speculative, variations of the abstract machine are described and discussed. These variations await practical evaluation.

6.9.1 Eager vs. Lazy Waking

The execution model and the abstract machine presented employ *eager waking* in that the number of steps before waking is bounded. AGENTS 0.9 employs a *lazy waking* scheme. Waking takes place in the guard instructions only, where it is useful to establish whether all (currently known) work in an and-box has been performed. Even lazier waking is conceivable, for example to wake only when all other work available has been performed.

The abstract machine could be changed for lazy waking as follows. Add a wake attribute to contexts, which is set to the top of the wake stack at choice-node entry, analogously to the other attributes. When waking, wake only nodes (and suspended calls) in the current context. Finally, wake only in guard instructions.

Observe that there is no need for resume or restore insertion point tasks in this scheme. There are also other advantages as well as disadvantages.

The advantage of lazy waking is mainly that a producer is not interrupted only because a consumer is interested. In the list-sum example of Chapter 2, the ea-

ger waking model presented here will exhibit “thrashing” behaviour when faced with a goal of the form

```
sum(L, S), list(N, L)
```

The sum call will suspend, since there is no list. The list call will produce one list cell, and then suspend to wake the sum call, which will consume the single list cell and then suspend, and so on. This will be extremely inefficient, at least an order of magnitude slower in a native coded implementation. A lazy waking model does not have this problem, since the producer would complete its task uninterrupted.

The advantage of eager waking is mainly that when there is interaction, a process will reply to a request early, thereby giving the client input to its subsequent work, for which a more expensive suspension might occur if input was not present. For example, if write(Term, InStream, OutStream) suspends unless InStream is known and waking is lazy, a process that first tries to obtain a stream from a server and then proceeds to write all the elements in a list will suspend one call for each element in the list, since the server will not respond eagerly.

Thus, lazy waking is, in some sense, more stable, but is not so intuitive since it can be expected that a process will reply to a request as soon as possible. This is related to fairness (discussed in Section 5.9.1). An experienced programmer can control eager waking to achieve the desired effects and avoid thrashing, but can also properly synchronise programs in the lazy waking scheme. Programs that are written to work well in the lazy scheme are “better” in that they can be expected to work reasonably well with many different execution models.

6.9.2 Incremental vs. Batch Aggregates

As described, unless the generating goal suspends, aggregates will stack up all collect statements before executing them in a “batch”. In the case of the example in Section 6.8.1, the list produced was not made available to consumers, which could have reduced the overhead of GC by consuming the list as it was produced. Incremental execution can be achieved as follows.

Introduce a new choice-task *resume aggregate*, with attributes

- *aggregate*: a reference to a choice-node
- *beyond*: a reference to a choice-task

To *collect* the current and-node, change its state to dead, unlink it from alternatives, and set the forward attribute to refer to the parent of the current choice-node. Then, push a resume aggregate task, with the current choice-node as aggregate, and the choice attribute of the current context as beyond. Mark the grandparent of the current and-node as unstable. Finally, pop the current context, set the current and-node to its grandparent, and decode instructions from the next instruction.

Note that the resume aggregate task is found when backing up, after failure or after suspension. Choice splitting is inhibited explicitly. The and-node is unstable, since the aggregate is not completely executed.

To *back up*, if it is a resume aggregate task, do as follows. If the current and-node (containing the aggregate) is dead, remove all choice-tasks to and including beyond, and back up. Otherwise, install the first alternative in the aggregate referred to, set choice in the current context to beyond, pop the task, and proceed.

This scheme does not in anyway interfere with other execution. It is quite possible to distinguish incremental aggregates from other aggregates, and compile the former using special incremental guard_collect instructions. All tasks pertaining to the and-node containing the aggregate will be executed, and thus also any consumers it contains. An obvious disadvantage is that it, quite unnecessarily, suspends this and-node before resuming the aggregate, and that it has to be marked as unstable. The ideal position for this task would be as the last and-task, but (1) this position cannot be reached, as it is part of a sequence of instructions preceding the guard, and (2) it would involve moving any number of preceding tasks.

A different scheme can be achieved by placing the resume aggregate task on the and-stack after the and-continuation corresponding to the collect statement. There is no “room” there, but the continuation could be moved, since it is likely to be small, or room can be reserved for the task “in” the continuation. By so doing, there is no need for marking the and-node as unstable, but only consumers suspended on the output of the aggregate will be given a chance to execute. This scheme also does not interfere with other execution.

6.9.3 Message-Oriented Scheduling

In the abstract machine presented, agents are compiled more as procedures that expect input and produce output, than as objects, the normal state of which is suspended and which receive intermittent communication.

There is a compilation scheme based on a different point of view called *message-oriented scheduling* [Ueda and Morita 1992]. If it is known that a message sent on a certain stream has only one recipient, the recipient and the stream can be compiled in such a manner that sending a message is analogous to invoking a method of an object. For FGHC, this knowledge requires modes, since it is not otherwise known where streams are produced and consumed. For streams connected to ports (see Chapter 7), it can be easy to know, since the stream is produced by the port.

Messages to port-based built-in objects in the AGENTS system are handled as method calls, which gives a flavour of eager waking. It is probably undesirable to mix different styles of waking, since this complicates the mental model for programmers. Thus, message-oriented scheduling should probably be combined with eager waking.

6.10 RELATED WORK

Much work has been done on abstract machines for Prolog, the committed choice languages, and some on combinations. The committed-choice languages are often flat, or, if the guards are deep, do not use hierarchical binding environments (constraint stores for tree equality constraints). We distinguish between *goal-based* models, that create explicit representations of all constituents of a composition (in AKL terminology), and *continuation-based* models, that create them implicitly via a continuation, as in the abstract machine described here.

Miyazaki, Takeuchi, and Chikayama [1985] describe a sequential interpreter for Concurrent Prolog, with deep guards and hierarchical binding environments. The computation state is a tree of AND- and OR-processes, which correspond to choice- and and-nodes. Their *shallow binding scheme* is very close the one employed here. The model is goal-based.

Crammond [1990] reports on a parallel abstract machine for PARLOG (the JAM) and a corresponding implementation, which supports deep guards and is comparable to SICStus Prolog in speed. However, as an optimisation, it allows only one active deep guard at a time. There may be either one deep guard in clauses composed by parallel clause composition, or any number of deep guards in clauses composed by sequential clause composition. The model is goal-based. Since PARLOG does not distinguish between local and external variables, there is no need to support a hierarchical binding environment.

The trend for the “pure” committed-choice languages is strongly converging to light-weight flat languages and implementations that compile to C, such as jc [Gudeman, De Bosschere, and Debray 1992] and KLIC [Chikayama, Fujise, and Yashiro 1993].

Yang [1987] describes a parallel implementation scheme for P-Prolog, a language offering a combination of concurrent execution and don't know nondeterminism, using determinism for synchronisation. A rather involved scheme for parallel execution is described which focuses on the treatment of bindings in a parallel context, allowing no direct comparison.

Santos Costa, Warren, and Yang [1991a; 1991b] present Andorra-I, a Prolog implementation based on the Basic Andorra Model (BAM, see Section 8.1.8) which provides both dependent-and and or-parallelism on the Sequent Symmetry. The model is goal based. Execution states are not hierarchical, but share linked lists of processes in an or-tree. These lists are semi-double-linked, a technique which, compared to full double-links, reduces the need for locking when manipulating lists in a parallel implementation, and reduces the need for trailing when shared nodes are updated. This technique is difficult to adapt to the present context due to its incremental construction and suspension of goals. (But other techniques have similar effects [Montelius and Ali 1994].)