Instant Replay Debugging of Concurrent Logic Programs

Kish Shen* Steve Gregory

Department of Computer Science
University of Bristol
Bristol BS8 1TR, U.K.

Abstract

One problem with debugging (committed choice) concurrent logic programs is that their behaviour may be non-deterministic, in that successive executions of the same program may produce different results. We describe a scheme, based on the 'Instant Replay' scheme developed for more conventional parallel languages, that allows us to reproduce the execution behaviour of a concurrent logic program on subsequent executions, so that the execution may be examined for debugging purposes. The properties of concurrent logic programming languages allow us to simplify our scheme greatly. We have demonstrated our scheme with KLIC, and KL1 on the PIM multiprocessors, but it can also be applied to other committed choice concurrent logic programming languages.

1 Introduction

One of the standard ways of debugging a sequential program is to re-execute the program when a bug occurs. This can be done because sequential execution is normally deterministic, so the bug would be reproduced. Moreover, a program's behaviour is not affected by the use of debugging tools, which can therefore be used to examine the execution in detail to try to understand the bug.

This re-execution technique has formed the basis of most approaches to debugging sequential logic programs, with various debugging tools provided to ease the examination and controlling of the execution (*e.g.* [4]). Indeed, even algorithmic debugging [10], which is generally regarded as very different from traditional methods of debugging logic programs, incorporates the idea of repeatedly executing a program or parts of a program, and examining the output in a systematic way.

For a concurrent language executing in parallel, the re-execution approach to debugging has some serious drawbacks, because the execution of a concurrent program can be non-deterministic. That is, the actual execution, and its results, can vary between runs, because it depends on the relative timing of the various processes that are executing concurrently. Thus, if a bug occurs in one execution, a re-execution of the program may not in general be able to reproduce the bug. Moreover, as the exact behaviour is often very time-critical, tracing the execution might greatly perturb the timings, making it even less likely that any non-deterministic bug will be reproducible.

Concurrent logic programming languages suffer from the problem of non-deterministic bugs, but this has generally not been taken into consideration in research work concerned with debugging (*e.g.* [6, 11]), although it has been acknowledged as a problem that needs to be tackled [6]. Tracing facilities of one form or another

*Current Address: Centre for Novel Computing, Department of Computer Science, University of Manchester, Manchester M13 9PL, U.K.

are usually provided with such systems (*e.g.* [3, 1, 12]), which allow the programmer to remove most bugs by executing the program sequentially. However, bugs which only manifest themselves when executed in parallel, and which are not consistently reproducible, are much more difficult to eliminate. In this paper, we describe a method, based on the 'Instant Replay' method proposed by LeBlanc and Mellor-Crummey [8], which allows this difficulty to be overcome.

1.1 Instant Replay

Instant Replay was proposed as a way to overcome the problems of debugging concurrent parallel programs. It was developed in the framework of conventional concurrent languages, but is applicable to most programming languages. The basic idea is to provide repeatable execution so that, when a program is being debugged, the same behaviour can be observed in as much detail and as often as desired. This is done by "recording" the execution of a program, and "replaying" the recorded execution should a bug occur. There is no risk of debugging tools perturbing the behaviour of the replay program, as its execution is controlled by the recording.

The important question is what needs to be recorded. At first sight, it might appear that a lot of information has to be recorded in order for the replay version to duplicate the behaviour of the recorded version: a concurrent program comprises processes which may be created and destroyed dynamically, and which are able to communicate with each other. It may seem that both the evolution of the processes and the detailed behaviour of each process have to be recorded. This would lead to a significant speed overhead, and the recorded execution would no longer closely resemble the original execution, increasing the probability of not reproducing some of the bugs in the unrecorded version. Another possible disadvantage is that the recorded information generated can be huge, and storage of such information problematic.

For the recording process to be as efficient as possible, the minimum of information should be recorded. This is the objective of Instant Replay. LeBlanc and Mellor-Crummey made the observation that, in most concurrent programs, the execution of a process is actually deterministic, *if* its inputs are known. Inputs to a process come as communications from other processes, or are supplied initially. This means that it is sufficient to know the order in which communications take place, *without* needing to know the contents of the actual messages that are transmitted: if the order is reproduced, the same communication will occur. This allowed the recording process to be much less expensive than in previous recording proposals, and so be usable in practice.

Instant Replay can be applied to a concurrent logic programming language to ease the problem of detecting non-deterministic bugs. Instant Replay will provide the ability to repeat equivalent executions of the program, and it can be integrated with existing tools for debugging the program.

2 Instant Replay for Concurrent Logic Programs: Basic Scheme

In this section, we describe how Instant Replay can be applied to committed choice concurrent logic programming languages. For concreteness, we use KL1 [14], specifically KLIC [2], as our target language, but the method is also applicable to other committed choice concurrent logic programming languages.

A straight adaptation of the Instant Replay method to a concurrent logic programming language would mean recording the order in which all variable bindings are made, as this is the way that processes (goals) communicate. However, committed choice concurrent logic programming languages have several properties that allow us to drastically simplify the recording process, compared to more conventional languages:

- Single assignment property: variables can only be assigned to once, and variables are the only means of communicating between different processes (goals).
- A variable cannot be assigned conflicting bindings without the whole program failing.

- ‘Don’t care’ non-determinism [7]: only one clause for a predicate is allowed to proceed beyond the guard.

These properties together mean that the only source of non-deterministic behaviour in a program which has user-visible effects is the clause to which each goal commits during execution. The other source of non-deterministic behaviour – the order in which goals within a clause is executed – does not usually have any observable effect on the computation. If the program succeeds, the same variable bindings are made. Deadlock, failure, and non-termination are also unaffected by the order of execution of goals. Thus, it is sufficient to record which clauses are committed to, and repeat these commitments during replay. Success (with the same answers), looping, failure, or deadlock in the recorded program will be faithfully repeated during replay. If the program contains more than one failing goal (outside of a guard), they may be replayed in a different order, which means that the record and replay executions will abort for different reasons. However, since each failing goal constitutes a bug, this should not be a problem.

Unlike the original or recorded executions, whose behaviour is time-dependent, the replay execution has no such dependence, and can thus be monitored in as much detail, and repeated as often, as desired.

2.1 Implementation of Basic Scheme

Although the recording can be done at a low level, it is easier to do it at the source level by a source-to-source transformation. The original program is transformed into two versions: one for record, and the other for replay. An extra argument is added to the head and every (non-built-in) goal of the program. This argument is a data structure, built by the record execution, that represents the commitments made by each goal. During replay, the structure from a previous record execution is used to control the execution so that the same commitments are made. As discussed, this is sufficient to ensure that the replay execution is equivalent to the recorded execution. The transformed programs (both record and replay) are referred to as being *monitored*.

As the transformation is done at the source level, only user-defined predicates can be modified and monitored in this way. Fortunately, most built-in predicates are deterministic in that, given the same input, they will produce the same result whenever they are executed, so there is no need to monitor them. We will discuss non-deterministic built-ins in more detail in Section 4.1.

The transformation is best illustrated by an example. Consider one clause from a KL1 program that finds the derivative of an expression:

```
d(U+V,X,D) :- D=DU+DV, d(U,X,DU), d(V,X,DV).
```

This clause is transformed into the following for recording:

```
d(U+V,X,D,_A) :- _A = c1(_B,_C), D=DU+DV, d(U,X,DU,_B), d(V,X,DV,_C).
```

Extra arguments are added to the head of the clause, and each non-built-in goal. An extra goal, `_A = c1(_B,_C)`, is added to the clause body to collect the structure built by goals in the clause, and pass it via the head argument (`_A`) to the caller of this predicate. The name of the structure being built (`c1` in this case) differs from the other clauses in this predicate, to identify the clause that is being committed to. Thus, a tree structure is built, each goal adding a node that identifies which clause is committed to. We refer to this data structure as the *trace tree*.

The trace tree generated by the record program is used by the replay program to control its execution. An extra argument is added to the head of each clause in the replay version of the program, forcing commitment to the same clauses as the recorded program used. In the replay transformation of the example clause, the extra argument (`c1(_B,_C)`) makes this clause the only one that will match a `c1` node in the trace tree:

```
d(U+V,X,D,c1(_B,_C)) :- D=DU+DV, d(U,X,DU,_B), d(V,X,DV,_C).
```

Figure 1 shows a more complete example of the record and replay transformation: it shows the whole derivative program, along with the record and replay versions of the same program.

Since the trace tree argument completely determines the clause used to reduce each goal in the replay version, the guard can be removed from the replay version of each clause if it is flat and produces no bindings for variables that appear in the clause body. Likewise, any control annotations for controlling the clause selection, such as the `alternatively` and `otherwise` keywords, can be removed.

Our scheme, as described above, is not restricted to languages with flat (built-in) guards. If a clause guard contains user-defined goals, the trace tree built by these guard goals forms part of the trace tree returned by the clause head; it is used in the replay version to reproduce the behaviour of the guard of the chosen clause. However, any computation that was performed by unsuccessful guards will not be recorded in the trace tree.

In the next section we shall assume that all built-in predicates are "observably deterministic", as defined below, though we consider relaxing this restriction in Section 4.1.

Definition: observably deterministic goal A goal is *observably deterministic* if, for a given input substitution, it will always either succeed with the same output substitution, or fail, or suspend.

3 Our Implementation: Improving the Basic Scheme

Our actual implementation contains several important enhancements over the basic scheme outlined in the previous section, including methods of reducing the size of the trace tree, and the handling of abnormal terminations. We now describe these improvements, along with some performance results of our implementation.

3.1 Removing deterministic nodes from trace tree

In the basic scheme outlined above, every call to a user-defined predicate is recorded. This means that a large program can generate a very large trace tree. Although the speed overhead of constructing the tree is small, the tree has to be kept in memory and, if it becomes sufficiently large, speed overheads will be incurred because of extra swapping/paging and garbage collection. Moreover, this overhead might significantly affect the behaviour of the recorded program, so that it no longer closely resembles the original program. In the worst case, the record program may not be executable at all. It is thus desirable to reduce the size of the trace tree as much as possible.

One way to do this is by exploiting the fact that some goals are deterministic: their behaviour will be reproduced during replay without the need for monitoring. By performing compile-time analysis to automatically detect 'deterministic' goals, this information can be used to avoid converting such 'deterministic' goals for record or replay. We shall now describe this compile-time analysis.

3.1.1 Definitions

Candidate clause A *candidate clause* for goal Q is a clause whose head successfully matches with Q and whose guard can be successfully executed.

Locally deterministic goal

- A goal Q for a built-in predicate is *locally deterministic* if it is "observably deterministic", as defined in Section 2.
- A goal Q for a user-defined predicate is *locally deterministic* if it never has more than one candidate clause.

```
% original

test :- d((x+1)*(x^2+2)*(x^3+3),x,D).

d(U+V,X,D) :- D=DU+DV, d(U,X,DU), d(V,X,DV).
d(U-V,X,D) :- D=DU-DV, d(U,X,DU), d(V,X,DV).
d(U*V,X,D) :- D=DU*V+U*DV, d(U,X,DU), d(V,X,DV).
d(U/V,X,D) :- D=(DU*V-U*DV)/V^2, d(U,X,DU), d(V,X,DV).
d(U^N,X,D) :- integer(N) | N1 := N-1, D=DU*N*U^N1, d(U,X,DU).
d(exp(U),X,D) :- D=exp(U)*DU, d(U,X,DU).
d(log(U),X,D) :- D=DU/U, d(U,X,DU).
d(X,X,D) :- D=1.
otherwise.
d(C,_X,D) :- D=0.

%record
test :- test(_A), builtin:print(_A).

test(_A) :- _A = c1(_B), d((x+1)*(x^2+2)*(x^3+3),x,D, _B).

d(U+V,X,D,_A) :- _A = c1(_B,_C), D=DU+DV, d(U,X,DU,_B), d(V,X,DV,_C).
d(U-V,X,D,_A) :- _A = c2(_B,_C), D=DU-DV, d(U,X,DU,_B), d(V,X,DV,_C).
d(U*V,X,D,_A) :- _A = c3(_B,_C), D=DU*V+U*DV, d(U,X,DU,_B), d(V,X,DV,_C).
d(U/V,X,D,_A) :- _A = c4(_B,_C), D=(DU*V-U*DV)/V^2, d(U,X,DU,_B),
d(V,X,DV,_C).
d(U^N,X,D,_A) :- integer(N) | _A = c5(_B) N1 := N-1, D=DU*N*U^N1,
d(U,X,DU,_B).
d(exp(U),X,D,_A) :- _A = c6(_B), D=exp(U)*DU, d(U,X,DU,_B).
d(log(U),X,D,_A) :- _A = c7(_B), D=DU/U, d(U,X,DU,_B).
d(X,X,D,_A) :- _A = c8, D=1.
otherwise.
d(C,_X,D,_A) :- _A = c9, D=0.

%replay
test :- builtin:read(_A), test(_A).

test(c1(_B)) :- d((x+1)*(x^2+2)*(x^3+3),x,D, _B).

d(U+V,X,D,c1(_B,_C)) :- D=DU+DV, d(U,X,DU,_B), d(V,X,DV,_C).
d(U-V,X,D,c2(_B,_C)) :- D=DU-DV, d(U,X,DU,_B), d(V,X,DV,_C).
d(U*V,X,D,c3(_B,_C)) :- D=DU*V+U*DV, d(U,X,DU,_B), d(V,X,DV,_C).
d(U/V,X,D,c4(_B,_C)) :- D=(DU*V-U*DV)/V^2, d(U,X,DU,_B), d(V,X,DV,_C).
d(U^N,X,D,c5(_B)) :- N1 := N-1, D=DU*N*U^N1, d(U,X,DU,_B).
d(exp(U),X,D,c6(_B)) :- D=exp(U)*DU, d(U,X,DU,_B).
d(log(U),X,D,c7(_B)) :-  D=DU/U, d(U,X,DU,_B).
d(X,X,D,c8) :- D=1.
otherwise.
d(C,_X,D,c9) :- D=0.
```

Figure 1: Example record and replay versions of a KL1 program

A candidate clause for a locally deterministic goal is uniquely determined by its arguments, so there is no need to record which clause is used. However, the goal may still need to be monitored (an extra argument added), because it may call other goals (directly or indirectly) which are not locally deterministic.

Totally deterministic goal

- A goal Q for a built-in predicate is *totally deterministic* if and only if it is locally deterministic.
- A goal Q for a user-defined predicate is *totally deterministic* if and only if Q is locally deterministic *and* every guard and body goal in every candidate clause for Q is totally deterministic:

  $$Q \in T \iff Q \in LD \wedge (\forall (H_m \ :- \ G_m | B_m) \in \mathit{def}(Q) : ((\forall g \in G_m \Rightarrow g \in T) \wedge (\forall b \in B_m \Rightarrow b \in T)))$$

  where

  LD = the set of locally deterministic goals,

  T = the set of totally deterministic goals,

  def(Q) = the set of clauses in the procedure for the predicate of goal Q.

Only goals that are not totally deterministic need to be converted for record and replay. (Recall that we are assuming for now that all built-in goals are totally deterministic; but see Section 4.1.)

3.1.2 Detecting determinism

To discover which goals in a program are totally deterministic, we perform a simple abstract interpretation of the program. We define the following abstract domain, arranged as a complete lattice as shown in Figure 2:

$$D_\alpha = \{\perp, [local\ det], [total\ det], \top\}$$

The concretisation function is:
$\gamma : D_\alpha \longrightarrow G, G =$ set of goals appearing in clauses of the program

$$\begin{aligned} \gamma(\perp) &= \emptyset \\ \gamma([local\ det]) &= \{locally\ deterministic\ goals\} \\ \gamma([total\ det]) &= \{totally\ deterministic\ goals\} \\ \gamma(\top) &= G \end{aligned}$$

The abstraction function α can easily be derived from the concretisation function. Note that $\top$ is the most general abstraction, as any goal can be abstracted to $\top$. The aim of the abstract interpretation is to classify as many goals as possible as $[total\ det]$, with the rest of the goals classified as $\top$.

The abstract interpretation is done as a two stage bottom-up analysis. First, all body goals occurring in the program are assumed to be locally deterministic, and are placed in $[local\ det]$. A local determinism checker is then used to check these goals: those that cannot be shown to be locally deterministic are moved to $\top$ in the abstract domain. The goals remaining in $[local\ det]$ are now known to be locally deterministic.

Next, the goals remaining in $[local\ det]$ are assumed to be totally deterministic and are placed in $[total\ det]$. These goals are checked to see if they really are totally deterministic; if not, they are moved to $\top$. The total

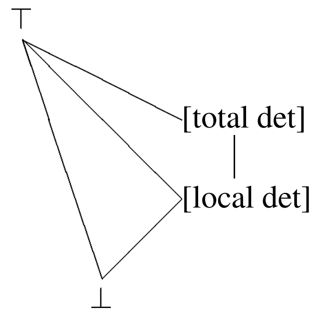

Figure 2: Determinism domain

determinism is checked by promoting the $\top$ status, so that the callers of goals which are abstracted to $\top$ are also moved to $\top$. That is, any goal that is assumed to be totally deterministic, but whose clauses contain any goals which are abstracted to $\top$, are also moved to $\top$. This is because they call goals which might be non-totally deterministic, and thus they might be non-totally deterministic themselves.

Fixed-point is reached when all goals that call goals abstracted to $\top$ are also abstracted to $\top$. When this condition is satisfied, all goals remaining in $[total\ det]$ can only call goals that are also abstracted to $[total\ det]$. Because the local determinism analysis will previously have moved to $\top$ any goals not known to be locally deterministic, the goals that remain in $[total\ det]$ must satisfy both conditions for total determinism, and are therefore proven to be totally deterministic.

In more detail, the algorithm for finding non-totally deterministic goals (by promoting such goals from $[total\ det]$ to $\top$ in the abstract domain) is as follows:

1. Initially, all goals already in $\top$ are placed in $\mathcal{R}$ (the set of goals remaining to be examined).

2. A goal, Q, is chosen and removed from $\mathcal{R}$. Suppose that the predicate of the head literal of the clause in which the goal occurs is P. All goals for P are then found. Such goals can currently either be in $[total\ det]$ or in $\top$ in the abstract domain. For any such goal, if it is in $[total\ det]$, it is moved to $\top$ and added to $\mathcal{R}$. The move to $\top$ is necessary because the goal cannot be totally deterministic: its clauses contain at least one goal, namely Q, that is not totally deterministic.

3. Step 2 is repeated until $\mathcal{R}$ is empty: fixed-point has been reached. As the algorithm is bottom up, starting from goals not known to be locally deterministic and then finding their callers, each goal is only examined once. As the number of goals is finite and usually small, the cost should be low. This has been confirmed by our experimental measurements in Figure 8.

We illustrate the algorithm for this stage with a simple example. Consider the (original) program in Figure 1. A local determinism analyser should be able to show that the `d((x+1)*...)` goal called by `test/0` is locally deterministic, because its first argument is a structure, and its second argument is an atom. The other `d/3` goals, however, might not be found to be locally deterministic, because their second argument is a variable, and can thus potentially match more than one clause of `d/3`. At the end of the local determinism analysis, all `d/3` goals except `d((x+1)*...)` would be placed in $\top$. Initially $\mathcal{R}$ would contain all `d/3` goals except `d((x+1)*...)`.

The second stage of the abstract interpretation can now begin: a goal from $\mathcal{R}$ is chosen and examined. As only the `d/3` goals are in $\mathcal{R}$, one of them will be examined. In this case it does not matter which, as they all have `d/3` as their caller. Thus, `d((x+1)*...)` is moved to $\top$ and also placed in $\mathcal{R}$. Now if any other `d/3` goal besides `d((x+1)*...)` is chosen from $\mathcal{R}$, no further action will be taken, as the caller is again `d/3`. However, when `d((x+1)*...)` is chosen, its caller is found to be `test/0`, which is not called by any goals. Thus, eventually all goals will have been examined, and $\mathcal{R}$ would be empty. At this point, the algorithm has finished, leaving all goals in $\top$. The result is that no goals are found to be totally deterministic.

In the given program, all `d/3` goals have an atom as their second argument, so they are actually locally deterministic. A sufficiently sophisticated analyser might be able to detect this. In this case, the algorithm would find that `d/3` is also totally deterministic.

The determinism information is then used by the converter, so that any goals that are totally deterministic are not converted. As we shall show in our results, this can have a dramatic effect on the size of the trace tree.

In our current implementation, we do not treat each goal individually as suggested above. Instead, we perform our analysis at the predicate level: all goals for a particular predicate are treated in the same way. Thus, we use notions of determinism that apply at the predicate level, instead of at the level of goals.

3.1.3 Predicate-level determinism

Definitions

Locally deterministic predicate A predicate P is *locally deterministic* if and only if *all* goals for the predicate are locally deterministic:

$$P \in LDP \iff (\forall Q_n : \mathit{pred}(Q_n) = P) \Rightarrow Q_n \in LD$$

where
$LDP =$ set of locally deterministic predicates,
$LD =$ set of locally deterministic goals.
$pred(Q) =$ predicate of goal Q.

Totally deterministic predicate A predicate P is *totally deterministic* if and only if all goals for the predicate are totally deterministic:

$$P \in TP \iff (\forall Q_n : \mathit{pred}(Q_n) = P) \Rightarrow Q_n \in T$$

where
$TP =$ set of totally deterministic predicates,
$T =$ set of totally deterministic goals.

Impact of predicate-level determinism
This weaker notion of determinism leads to a somewhat less precise determinism analysis. This is because, if there exists a non-totally deterministic goal for a predicate (even if such a goal does not appear in the program), then *all* goals for the predicate are treated as non-totally deterministic. In the case of the program of Figure 1, the `d/3` predicate is not considered locally or totally deterministic, because it is possible for a goal for `d/3` to be non-deterministic (an example of such a goal is `d(1+1,1+1,D)`).

The predicate-level notion of determinism has little practical impact on the rest of the abstract interpretation process: the promotion of local determinism to total determinism. The meaning of the abstract domain values has to be somewhat altered, and the concretisation function changed to the following:

$$\gamma(\bot) \quad = \quad \emptyset$$

$$
\begin{aligned}
\gamma([local\ det]) &= \{locally\ deterministic\ predicates\} \\
\gamma([total\ det]) &= \{totally\ deterministic\ predicates\} \\
\gamma(\top) &= G
\end{aligned}
$$

but the actual algorithm used for the promotion stage remains unchanged. It is less costly because there are fewer items to be examined: predicates instead of individual goals are selected from $\mathcal{R}$ at each step.

Despite the loss of precision, there are many advantages to analysing at the predicate level: the analysis is much simpler, and need be done just once for each predicate, not for every different call to it. The abstract interpretation is also simpler. Moreover, it is nearly always true that, if one goal for predicate P is deterministic, then all goals for P are deterministic. The `d/3` example is the only counterexample that we have found.

Our analyser comprises two orthogonal components: checking a predicate for local determinism, and propagating this information to detect total determinism. The quality of the analysis, and therefore the trace tree size, depends largely on how well the system is able to detect which predicates are locally deterministic.

3.1.4 Local determinism analysis

The algorithm that we use to detect local determinism is adapted from the determinism analyser developed to analyse Prolog programs in the Andorra-I system [9], though the simpler nature of KL1 makes many simplifications possible. The details of the analyser are somewhat beyond the scope of this paper, but the main features are as follows:

- Clauses of a predicate are checked pairwise for mutual exclusiveness: the predicate is considered to be locally deterministic if all clause pairs are mutually exclusive. The algorithm is conservative: it will report that a pair is mutually exclusive only if it is mutually exclusive, but the algorithm may not detect *all* pairs which are actually mutually exclusive.
- Two clauses are mutually exclusive if their heads are not unifiable. Otherwise, the binding environment from unifying the clause heads is retained, and the guards of the clauses are checked.
- The constraints that the guards impose on common variables in the clause heads are considered. These constraints include types and ranges of numeric values. The two clauses are considered mutually exclusive in any of the following cases:
  - Range constraints in the two clauses are inconsistent. For example:

    ```
    p(X) :- X < 0 | b1(X).
    p(X) :- 0 < X | b2(X).
    ```

  - Type constraints in the two clauses are inconsistent. For example:

    ```
    p(X) :- atom(X) | b1(X).
    p(X) :- list(X) | b2(X).
    ```

  - The constraints in one clause conflict with bindings in the other. For example:

    ```
    p(0) :- b(X).
    p(X) :- 0 < X | b2(X).
    ```

- In KL1, clauses of a predicate may be divided into groups separated by the `otherwise` keyword: a group of clauses is tried only when all clauses of preceding groups have failed. This effectively makes the groups of clauses mutually exclusive. Thus, only clauses *within* each group need to be mutually exclusive for the predicate to be locally deterministic. Our analyser takes this into account by performing determinism analysis only between pairs of clauses from the same group. If all groups are locally deterministic, the predicate is locally deterministic.

Our current analyser does not retain information between groups: it is possible for two clauses in a group to be non-exclusive when considered in isolation, but mutually exclusive if the failure of a preceding group of clauses is taken into account. Fortunately, such situations seem to be rare, but an example is:

```
a(0) :- b(X).
otherwise.
a(X) :- X =< 0 | b1(X).
a(X) :- 0 =< X | b2(X).
```

- The `alternatively` keyword, used in KL1 to give priority to some clauses over others, is ignored in the analysis because it does not affect local determinism.

The algorithm outlined above is not guaranteed to detect every predicate that is locally deterministic according to our definition (the `otherwise` example above is one case that we do not detect). Such a guarantee is impossible in the general case because it may involve solving arbitrarily complex problems. However, in practice, our algorithm has proven to be very effective in detecting local non-determinism. We tested the analyser on the 20 example programs that are supplied with the KLIC system, plus three larger KLIC programs available to us: `tsp`, a program to solve the travelling salesman problem, originally written to perform experiments with speculative parallelism in Parlog [5]; `chukl` [15], a finite-domain constraint handling system, and `convert`, the KLIC version of our converter, translated from the original Prolog program. The results are shown in Figure 3. The 'name' column is the name of the program, 'lines' is the number of lines the source program has, including any comments, 'pred.' is the number of different predicates that the program has, and 'non-det.' is the number of non-locally deterministic predicates remaining after the analysis.

name	lines	pred.	non-det.	name	lines	pred.	non-det.
deriv	33	4	1	primes	25	8	0
fact	8	2	0	primesp	25	8	0
hanoi	40	5	0	prio	12	4	0
iotest	9	2	0	puzzle	327	29	0
kkqueen	63	8	0	qlay	56	9	2
life	195	16	0	qsort	22	5	1
mastermind	127	20	0	turtles	115	20	0
merge	20	5	0	vecstr	17	2	0
parsetest	29	5	0	wave	184	30	3
pascal	79	11	0	tsp	494	59	1
pp	15	4	0	chukl	2383	151	8
				convert	2569	217	12

Figure 3: Results of Determinism Analysis

The results show that the analyser is able to determine that most predicates are locally deterministic. This suggests that the current analyser is indeed able to effectively detect determinism in KLIC programs, despite performing the analysis only at the predicate level.

An example of the benefit of performing the determinism analysis is shown in Figure 4. This is a fragment of a program for solving the N-Queens puzzle. With the basic scheme, the record version of the program would

```
% original
go(N,M) :-
    gen(N,L),
    queen(L,[],[],X,[]),
    count(X, M).

%record 1
go(N,M,_A) :-
   _A = c1(_B,_C,_D),
    gen(N,L, _B),
    queen(L,[],[],X,[], _C),
    count(X, M, _D).

%record 2
go(N,M,_A) :-
    _A = c1(_B),
    gen(N,L),
    queen(L,[],[],X,[], _B),
    count(X, M).
```

Figure 4: Benefit of determinism analysis

be converted as shown in `record 1` (the replay conversion would be similar). With determinism analysis, if the analysis is able to determine that both `gen/2` and `count/2` are totally deterministic, then these two predicates do not have to be converted, and the record conversion would be as shown in `record 2`.[1]

3.2 User declaration of determinism

As well as the automatic detection of locally deterministic predicates, we also allow the programmer to specify that a predicate (`<name>/<arity>` in module `<module>`) is locally deterministic, by means of a declaration

```
:- deterministic <module>:<name>/<arity>.
```

This has several applications:

- To help the analyser in cases where it is unable to detect locally deterministic predicates automatically; for example, the `a/1` predicate in Section 3.1.4 can be declared locally deterministic. No matter how sophisticated the analyser, there may always be cases where it cannot detect the determinism, so this facility can always be useful.
- As mentioned above, our current analyser is not sophisticated enough to detect the determinism of individual goals. An example is the derivative program of Figure 1. All calls to `d/3` appearing in the program are locally deterministic because their second argument is an atom, but the predicate itself is not. To avoid unnecessary monitoring of these goals, the programmer can state that the predicate `d/3` is locally deterministic.
- There is a kind of non-determinism that is very common in concurrent logic programs, whereby several candidate clauses exist but their body goals (if any) are identical. We shall call this *false non-determinism*. An example of this is the minimum relation, `min(X,Y,M)`, which returns `M` as the minimum of `X` and `Y`:

[1] In our actual scheme, no conversion is needed, because the analyser is able to detect that `queen/5` is also totally deterministic.

```
min(X,Y,M) :- X =< Y | M = X.
min(X,Y,M) :- Y =< X | M = Y.
```

If X and Y have the same value, there are two candidate clauses but their body goals are the same. The programmer can avoid unnecessary monitoring by declaring `min/3` deterministic.

- Another use for user-annotated local determinism is to selectively turn off the monitoring of parts of a program in a controlled way during program development. If a predicate that is not totally deterministic is believed to be "bug-free", then there is no need to monitor it provided that it is "observably deterministic": its inputs completely determine its eventual outcome. In other words, we effectively treat the predicate as a "black box", ignoring any internal non-determinism it might have.

  An example of a predicate that is observably deterministic but not totally deterministic is `qsort` (Figure 5): the `partition` goal is (rather gratuitously) non-deterministic, and therefore so is `qsort`. However, the output of `qsort` is a function of its input. Another example is the summing of the numbers in a list produced by the non-deterministic merging of two lists of numbers (Figure 6). The `merge` predicate is non-deterministic, and therefore so is `sumlists`, but its output is a function of its inputs.

```
qsort([X|L],R,R0) :-
   partition(L,X,L1,L2), qsort(L2,R1,R0), qsort(L1,R,[X|R1]).
qsort([],R,R0) :- R = R0.

partition([X|L],Y,XL1,L2) :- X =< Y | XL1 = [X|L1], partition(L,Y,L1,L2).
partition([X|L],Y,L1,XL2) :- X >= Y | XL2 = [X|L2], partition(L,Y,L1,L2).
partition([],_,L1,L2) :- L1 = [], L2 = [].
```

Figure 5: Quicksort program

```
sumlists(L1,L2,Sum) :- merge(L1,L2,L), sumlist(L,0,Sum).

sumlist([],Total,Sum) :- Total = Sum.
sumlist([N|Ns],Total,Sum) :-
   Total1 is Total+N,
   sumlist(Ns,Total1,Sum).
```

Figure 6: Sum of lists program

  Recording can be turned off by specifying that all predicates called are locally deterministic, even if they are not. For the `qsort` example, this means declaring `partition` as locally deterministic (`qsort` is locally deterministic anyway).

With the help of determinism declarations to specify those predicates that are locally deterministic, or falsely non-deterministic, all of the example programs supplied with the KLIC system can be made totally deterministic; only one predicate is left non-locally deterministic in `tsp` and `chukl`, namely `merge`.

3.3 Removing redundant nodes in trace tree

Another way to reduce the size of the trace tree is by removing redundant nodes. Normally, if a clause is a member of a predicate which is locally deterministic but contains body goals that are not totally deterministic,

```
go(N,M,_B) :-
    gen(N,L),
    queen(L,[],[],X,[], _B),
    count(X, M).
```

Figure 7: Example conversion with removal of redundant nodes

then the clause must be recorded. However, if only one of the body goals is not totally deterministic, there is no reason to monitor the commitment itself, although the non-totally deterministic body goal still has to be monitored. No trace tree node needs to be created for the clause: instead, the extra argument is just passed to the non-totally deterministic body goal.

This improvement can be illustrated using the N-Queens fragment in Figure 4. `go/2` is locally deterministic, as it has only one clause. It also contains only one goal, `queen/5`, that has to be monitored, so the local node can be removed, resulting in the converted code shown in Figure 7 (for both record and replay versions).

The trace tree size can be reduced in this way whether or not determinism analysis is done. However, determinism analysis reduces the number of goals being monitored, making it more likely that only one goal in the body remains to be monitored. In fact, many KL1 programs are largely deterministic, with only a few local areas of non-determinism. If these non-deterministic goals are called after a long chain of otherwise deterministic goals, this optimisation removes many intervening redundant nodes.

Another, less significant, optimisation can be applied to the case where a locally deterministic goal has two body goals which are not totally deterministic. Here, the structure for holding the extra argument can be changed from a structure like `c1(A,B)` to a list `[A|B]`. Under the common way of representing structures and lists, the structure representation requires three words whereas the list representation requires only two.

3.4 Abnormal termination

The trace tree is generated during the execution of the record program and is written to a file when the program terminates. An obvious way to do this is by adding a suitable output goal to the query, which is invoked when the main query has succeeded. This works well in normal conditions, for programs that terminate successfully, but many programs are intended to run indefinitely. Moreover, bugs in programs (such as infinite loops) may cause non-termination, while other bugs may cause a program to terminate abnormally. Abnormal termination can result from failure or deadlock (or, in the case of KLIC, perpetual suspension: a condition that will lead ultimately to deadlock).

The problem is that, although the trace tree is always generated, there is no direct way to make it available if the program does not terminate normally. To solve this problem, KLIC has recently been extended with a facility, *postmortem processing*, whereby a special goal (designated by the programmer) is called after both normal and abnormal termination. In the record version of a program, we use postmortem processing to call a goal that writes the trace tree to a file. This directly solves the problem of abnormal termination. If the programmer suspects that a program has gone into an infinite loop, or if unexpected behaviour is observed for an intentionally non-terminating program, the program can be interrupted to cause an abnormal termination and output the trace tree. The execution can then be replayed to examine for errors and possible infinite loops.

If a program terminates normally, the trace tree produced will be a ground term. However, with abnormal termination, it may contain unbound variables. As the postmortem processing is invoked *after* the rest of the program has finished execution, we can be sure that these unbound variables in the trace tree represent goals that have not reduced at the time when the program was terminated. A trace tree containing unbound variables will be read in unchanged for use by the replay version of the program. Clearly, a goal that did not reduce in the recorded execution will not reduce when replayed, because its node in the trace tree will be an unbound variable. However, it is not immediately obvious that this non-ground trace tree does faithfully reproduce the recorded program's behaviour. An unbound node can be generated by the record version under three conditions:

- The unbound node represents a non-totally deterministic goal that has failed. In this case, the goal will also fail during the replay execution, *provided* that the "real" arguments are matched before the trace tree argument. If the trace tree argument were matched first, the goal would suspend on it and deadlock or (in KLIC) perpetual suspension would occur instead of failure.

  In KLIC we can ensure that the trace tree argument is matched last by matching the argument by a unification goal at the end of the guard, not in the clause head. This is what is actually done in our converter. For example, the first clause of our `d/3` predicate of Figure 1 is translated for replay to:

  ```
  d(U+V,X,D,_A) :- _A = c1(_B,_C) | D=DU+DV, d(U,X,DU,_B), d(V,X,DV,_C).
  ```

- The unbound node represents a suspended goal: one for which all clauses are suspended. During the replay execution, the goal will also suspend. By matching the trace tree argument last, we can be sure that the goal will suspend on variables in the "real" arguments, not on the variable in the trace tree.

- The computation was interrupted and the unbound node represents a goal that was unexecuted at this time. In this case, during the replay execution, the goal will suspend on the variable in the trace tree and either deadlock or perpetual suspension will be detected (depending on the features of the host system). It is preferable that perpetual suspension is *not* detected, as this would cause the execution to abort early. If the execution is allowed to continue until deadlock, the whole of the computation that was performed before the interruption will be reproduced.

3.5 Cost of Performing the Conversion

It is expected that the analysis and conversion of the program will be done each time the program is compiled during its development so, the faster this process, the more practical the scheme will be.

The cost of performing the conversion comprises three main parts:

- The local determinism analysis. This includes the cost of reading the program in and, in our current implementation, the cost of asserting a representation of the input program onto the database. The actual cost of the analysis is complex and depends on many factors, such as how many predicates are deterministic, how difficult it is to check the exclusiveness of each pair of clauses, and so on.

- The promotion of the determinism information. This cost should be small and is proportional to the length of the program, as the predicates in the program only have to be scanned at most once.

- The actual conversion of the program. This includes the cost of writing out the converted programs (record and replay). Again, the cost is proportional to the length of the program, as each clause in the program has to be converted once for the record version and once for the replay version.

Figure 8 shows the time taken to perform the three phases of the conversion of the programs `tsp`, `chukl`, and `convert`, with the average and standard deviation of five timings shown for each measurement. Timings for the other programs of Figure 3 are not shown because they were too short for accurate measurement. The timings are obtained with the Prolog version of the converter; the KLIC version (`convert`) runs at about half the speed of the Prolog version. The second column indicates the number of lines in the program being converted,

prog.	# lines	det.	prom.	conv.
tsp	494	0.352±0.004	0.004±0.005	0.350±0.018
chukl	2383	1.296±0.030	0.020±0.000	1.098±0.019
convert	2569	2.398±0.325	0.172±0.007	1.894±0.031

Figure 8: Timings for Conversions

the third column the time taken (in seconds) for the determinism analysis, the fourth column the time taken for the promotion of the non-determinism information, and the fifth column the time taken for the actual conversion of the programs. All measurements are taken from a SparcStation 10/41, with the converter running under SICStus Prolog version 2.1#9, with the program compiled to non-native code.

The results shows that the converter is able to convert 600 to 1000 lines of source code per second. In fact, most of the time is probably spent in I/O, reading the program in, and then writing the converted programs out. If this conversion were done as part of the compilation process, then the I/O cost would be reduced, as the program would already be read in.

3.6 Expected Method of Use

With the availability of Instant Replay we expect that, during the initial stages of development of a program, all non-deterministic goals are recorded. Then, as the programmer becomes confident that particular sections of the code are bug free, recording is turned off by declaring those predicates locally deterministic. Eventually, no recording is done at all in the 'production' version. As long as the cost of recording is low, we expect to be able to catch most time-dependent (and usually non-repeatable) bugs during development. Naturally, the program used to replay a recorded execution must be consistent with the program that recorded it: both must be generated from the same source program with the same level of determinism declarations, otherwise the trace tree will produce the wrong results.

When the determinism analysis is performed, the user can examine which goals the program thinks are not locally deterministic, and annotate any such goals which are actually locally deterministic. Examining the result of the determinism analysis has another advantage: it allows the user to eliminate unintentional non-determinism. Our analyser even allows the programmer to check for non-determinism *instead* of performing the conversion. For this use, the non-determinism check continues even after non-determinism is detected, so that all non-exclusive pairs of clauses are reported to the user. The user can then examine each of these pairs of clauses to see whether the non-exclusiveness is intended.

In conjunction with other tools, we expect that Instant Replay can help with "performance debugging". The execution of a program is recorded, and then replayed and observed to see how performance can be improved.

4 Other Issues

4.1 Non-deterministic built-ins

4.1.1 Non-deterministic built-ins in clause bodies

One assumption made in our above approach is that built-in predicates are deterministic: their output, or whether they succeed or fail, are determined solely by their input arguments. This is usually the case, but there are a few exceptions. First we consider several ways to reproduce the result of non-deterministic built-ins appearing in clause bodies:

- **Defining by multiple clauses**. Replace the built-in goals by calls to suitable user-defined predicates that have several clauses, and then convert these procedures for record and replay in the usual manner.

  A KLIC built-in that can be handled in this way is `unbound(X,Result)`, which produces a different binding for `Result` depending on whether `X` is bound or unbound. `unbound(X,Result)` could be defined by the procedure in Figure 9. The first clause returns `Result=no` if `X` is bound (because the guard `wait(X)` succeeds). If `X` is unbound, `wait(X)` will suspend but the second clause will be used to return `Result=yes`. (Clauses following an `alternatively` keyword are tried only if the preceding clauses have all suspended or failed.) The `unbound` procedure shown is then converted to record and replay versions in the usual way.

```
unbound(X,Result) :- wait(X) | Result = no.
alternatively.
unbound(X,Result) :- Result = yes.
```

Figure 9: The `unbound/2` built-in as a procedure

- **Storing input arguments**. Arrange for the record version of the program to make a copy of the input arguments of the goal (replacing each variable by a new variable) *before* executing it. The copied arguments are stored in the trace tree, and retrieved at replay time before the goal is re-executed.

  This technique can be used to reproduce the behaviour of a built-in whose result depends on the binding state of its arguments. One example of such a built-in is a `write` predicate that immediately prints its argument term even if it contains variables. Another is the above `unbound/2` built-in, which can be defined as in Figure 10. The first (input) argument of `unbound/2` is copied using the `copy_term/2` built-in: this copies its first argument to its second, with no suspension.

```
% record
unbound(X,R,_A) :- copy_term(X,_A), builtin:unbound(_A,R).

% replay
unbound(X,R,_A) :- builtin:unbound(_A,R).
```

Figure 10: The `unbound/2` built-in converted to store input arguments

- **Storing output arguments**. Arrange for the record version of the program to store the output arguments of the goal in the trace tree *after* it is executed. Then, at replay time, the goal is not executed: its output arguments are simply unified with those stored in the trace tree.

  This technique is suitable for any built-in that does no side-effects and produces ground outputs. For example, the `unbound/2` built-in can be handled in this way, as in Figure 11. Here, the second (output) argument is stored in the trace tree.

  A more interesting application of this technique is to reproduce the behaviour of built-ins that interact with the program's environment. Recording the actual values supplied by the environment when the execution is recorded is the only way to reproduce the behaviour at replay time. For example, the `read/1` built-in can be converted as in Figure 12.

  Another possible application of this technique is to reproduce the behaviour of the built-in non-deterministic `merge` predicate, in the special case where the streams being merged are ground.

```
% record
unbound(X,R,_A) :- _A = R, builtin:unbound(X,R).

% replay
unbound(X,R,_A) :- _A = R.
```

Figure 11: The `unbound/2` built-in converted to store output arguments

```
%record
read(X, _A) :- _A = X, builtin:read(X).

%replay
read(X, _A) :- _A = X.
```

Figure 12: The `read/1` built-in converted to store output arguments

- **Special-purpose techniques**. The built-in merge predicate of KLIC can be handled by one of the above methods. However, since `merge` tends to be heavily used, and since the built-in `merge` is so much more efficient than a user-defined one, it would be desirable to find an alternative way to record the execution of the built-in `merge` itself. Perhaps the best solution is one suggested by Ichiyoshi[2]: to record the order in which the input streams supplied the elements to the merged stream. This method appears to be faster and more space-efficient than the method outlined above, though the recording will still impose some overhead compared with the unrecorded `merge`.

The built-ins that cannot easily be handled are those whose effect depends on their *order of execution*. These include goals that perform side-effects, such as some Input/Output built-ins and any form of destructive assignment. There is no way to preserve the order of execution of such goals without resorting to the techniques used for Instant Replay execution of conventional imperative languages. However, there are many well-known reasons to avoid the use of side-effects in concurrent logic programs, so this problem should not arise. For example, in KLIC, all I/O is performed by sending messages to a single process that actually performs the I/O.

4.1.2 Non-deterministic built-ins in guards

For a language that allows "deep" guards, the execution of a guard goal is recorded and replayed in the same way as a body goal, and any of the above techniques can be applied to handle non-deterministic built-ins called (directly or indirectly) from guards.

A flat language like KL1 restricts guards to contain built-in predicates, usually a subset of those provided in the language. It is unlikely that non-deterministic built-ins will appear in guards. However, if they are allowed, their results must be reproduced on replay. Perhaps the best way to do this is by a variant of the "storing output arguments" technique: the record version would store, in the trace tree node for goal Q, the values of any output arguments of the guard goals (assuming that these are ground) that are used in the clause body. When the clause is selected at replay time, these values are retrieved from the trace tree.

[2] Personal communication with Nobuyuki Ichiyoshi, 1994.

4.1.3 Discussion

The technique of storing output arguments in the trace tree can be used in other situations. For example, it can be applied to "turn off" the monitoring of parts of a program that are not "observably deterministic", but which the user is confident about. During the record execution, the outputs of such a goal are recorded in the appropriate trace tree node; during replay, the recorded output is used instead of executing the goal. The advantage of this approach is that it can reduce the recording overhead, and omits irrelevant execution details during the replay phase. A limitation is that the outputs of the goal must be ground.

4.2 Partial Transformation and Low-level Support

With determinism analysis, only some of the predicates of the program are transformed. It is possible that the name of a transformed predicate will clash with an existing untransformed predicate. The transformation program can deal with this by renaming one of these predicates if necessary.

As already discussed, in order not to record "observably deterministic" predicates that the programmer knows to be bug-free, it may be necessary to label some non-deterministic predicates as deterministic. If such predicates are used elsewhere in the program, two copies of the predicate (with appropriate name changes) have to be created, one labelled as "deterministic", the other not.

Currently, we perform the monitoring at a high level. Although, as we shall show, this is not in practice a very significant overhead, it will nevertheless be more efficient if the monitoring is more integrated with the KLIC system, and is done at a lower level, without doing a source-to-source transformation — for example, a more direct and compact representation of which clause has committed can be used than introducing an extra argument. An additional advantage of lower level monitoring is that built-ins can also have special support for monitoring, allowing for greater efficiency than the general methods discussed in Section 4.1.

5 Results

As a parallel version of KLIC is not yet available at the time of writing, our initial evaluation concentrated on the sequential version of KLIC. We have also been able to test the scheme and obtain results using KL1 on one of the Parallel Inference Machines at ICOT: PIM/M.

5.1 Results for KLIC

For a given program and query, the smaller the trace tree, the less the record version of the program is perturbed, and the closer it reflects the behaviour of the original program. We measure the size of the trace tree by simply counting the number of characters in its printed representation, a reasonably accurate and simple method of measuring relative sizes of terms.

As already shown in Figure 3, most of the example programs provided with the KLIC system are deterministic. That is, all their predicates are totally deterministic. With the help of user annotation, we are able to avoid converting any goals and, effectively, no trace tree is generated at all.[3] However, with the unoptimised basic scheme where all non-built-in goals are converted, the size of the tree generated can be substantial. For example, for the $5\times4\times3$ puzzle program [13] (a deterministic program), the size of the trace tree is just over a million characters, compared to a size of 0 if determinism analysis is done. The execution time of the record version is 3.5 ± 0.4 seconds, compared to 2.4 ± 0.6 seconds for the original unconverted program (both figures are averages and standard deviations for 3 timings). We expect that if monitoring is done at a lower level, where

[3] In our actual system, we do generate a 'stub' so that a one-node trace tree is generated.

the results are immediately written to a file (or files, one per processor), instead of being kept in memory, the overhead will be acceptable, even with the naïve monitoring approach.

For a more in-depth test, we tried the system on the two larger programs, `tsp` and `chukl`. The `tsp` program creates one or more solver processes to solve the travelling salesman problem by a parallel branch-and-bound search. With more than one solver process, the program can behave non-deterministically in that the search space can change from run to run. The `chukl` program uses non-deterministic merge in many places.

problem	ex. time	naive	non-ann.
6	0.027±0.005	18663	117
10	0.143±0.012	110425	309
15	0.497±0.029	414500	453
20	4.472±0.174	3289836	2917
25	8.266±0.090	7383687	3485
30	69.316±0.667	>60000000	14901

Figure 13: Measurements for Travelling Salesman Problem

The measurements we made for the KLIC version of the `tsp` program are summarised in Figure 13. The analyser is able to detect all the deterministic goals automatically, leaving only the `merge` predicate as locally non-deterministic, and so no user annotations are needed to aid the system. Each row of the table represents the results for one instance of the travelling salesman problem run using two solver processes. The first column identifies the problem: the number given is the number of cities that the salesman must visit, with the route cost matrix generated randomly. The second column gives the average execution times in seconds for each problem (without monitoring), with the standard deviations. Problems 6, 10 and 15 are averages of three timings, while 20, 25 and 30 are averages of 5 timings. The third column is the size of the trace tree (in number of characters) generated by the naïve record version of the program. The size for problem 30 is an estimate, as we were unable to store such a large trace tree. The fourth column gives the trace tree size using automatic determinism analysis, without user annotations, with the redundant node optimisation implemented. All the measurements are performed on a SparcStation 10/41. As can be seen, the analysis dramatically reduces the trace tree size.

If the program is run with only one solver process, then it is deterministic, and the versions with determinism analysis and annotations effectively generate an empty trace tree, whereas the naïve version would still generate trace trees of similar sizes to those it generated for two solver processes.

problem	original	record
25	8.266±0.090	8.354±0.154
30	69.316±0.667	69.572±0.880

Figure 14: Execution times for unmonitored and record versions of the travelling salesman program

We have not shown the execution time for the record version of the program in Figure 13, because we were not able to observe any meaningful differences in execution time between the original and record versions of the program for problems smaller than 25. Even for problems 25 and 30, whose timings for both the original and recorded versions are shown in Figure 14, the differences are well within the margin of error, with the difference being less than 0.5% for problem 30.

The results for the second program we examined, `chukl`, are not always as favourable as for `tsp`, even when there are few non-deterministic predicates. The framework for the constraint handling in `chukl` is written in KLIC, and both the size of the program, and the execution times, are substantial. For our measurements, we tested `chukl` solving the N-Queens problem using finite domain constraints. Of the eight predicates that our analyser was not able to determine to be locally deterministic, seven are either locally deterministic or are falsely non-deterministic, as described in Section 3.2; thus only the non-deterministic merge is a 'true' non-deterministic predicate.

problem	ex. time	naive	non-ann.	ann.
4	0.188±0.042	94837	30350	29027
5	0.528±0.019	313455	103814	99680
6	2.106±0.023	1285845	457877	440645
7	9.784±0.560	4796111	1794525	1735102

Figure 15: Measurements for CHUKL running N-Queens

problem	original	ann.
4	0.188±0.042	0.252±0.017
5	0.528±0.019	0.772±0.010
6	2.106±0.023	3.484±0.029
7	9.784±0.560	15.32±0.511

Figure 16: Execution times for unmonitored and record versions of the N-Queens program

We show the results for solving the N-Queens problem with N ranging from 4 to 7 in Figure 15, and compare the execution times for the record version (with user annotation) and the original program in Figure 16. The recording introduces an overhead of between 35 and 65%, using the annotated version of the program where only the non-deterministic merge is considered non-locally deterministic. The size of the trace tree is reduced by about three times from the naïve conversion by performing the determinism analysis. Moreover, although the user annotation is able to reduce the number of non-locally deterministic predicates from seven to one, the effect on the size of the trace tree is small: a reduction of only about 4%. The reason is that `merge`, the only non-locally deterministic predicate, is used extensively in different places in the program, so many predicates have to be monitored.

Even for an unfavourable example such as `chukl`, the overhead of 35 to 65% still does not dominate the 'real' computation, and thus monitoring is probably still useful, particular for runs of the program where the execution time is short. In practice, it is probably only such short executions that can be examined by any debugging tools in detail in any case. The behaviour of the monitored program is different from the original program, but it is probably still sufficiently close to reproduce many of the bugs that occur in the original program. Also, the overheads can be reduced by using one of the techniques to record the built-in non-deterministic merge described in Section 4.1. We believe our method is highly suitable for monitoring KL1 programs, as they typically exhibit a high degree of determinism. `tsp` seems to represent a program which is very suitable for our scheme, and `chukl` represents a less suitable case. The naïve conversion simulates the worst case situation where no goals are totally deterministic. Even in such cases, our data suggest that recording is still possible for programs that do not run for very long.

5.2 Results for KL1 on Parallel Inference Machine

The distributed version of KLIC is not yet available at the time of writing, but we were able to verify the parallel behaviour using the Parallel Inference Machines (PIMs) at ICOT. Our converter had to deal with the slightly different dialect of KL1 used on the PIMs.

The results for PIM/M, the distributed PIM architecture, are summarised in Figures 17 and 18. Figure 17 shows the measurements for the unconverted `tsp` program, and for the record version of this program, for problem 25. The number of solver processes used is 32, as this allowed up to 32 processors to be utilised.

# processors	original			record		
	# nodes	# reductions	time	# nodes	# reductions	time
1	263	2,061,349	12.70±0.09	263	2,061,349	12.56±0.22
16	546±59	4,884,595±362,376	8.36±0.84	571±80	5,057,071±644,027	8.17±1.45
32	716±64	6,042,033±386,002	10.64±0.75	633±93	5,364,125±575,348	8.11±2.05

Figure 17: Measurements on PIM/M for the original and record version of `tsp`

type	# nodes	# reductions
rec	558	4,922,442
1	558	4,919,265
16	558	4,922,674±99
32	558	4,922,407±80

Figure 18: Measurements on PIM/M for replay version of `tsp`

The measurements are for 1, 16 and 32 processors. For each configuration, we show the number of search tree nodes expanded, the number of reductions performed, and the execution time (in seconds). Each measurement is the average of three executions, with the standard deviation shown after the ± sign where appropriate. Note that our results do not show significant parallel speedups[4]; our aim is rather to test the Instant Replay scheme by maximising the non-deterministic behaviour.

The behaviour of the original and record versions of the program is very similar, with both the execution time and the number of reductions performed by the two versions being within the margin of error in all configurations. As expected, the behaviour of the program on one processor for both version is deterministic, as the number of nodes explored does not vary. For more than one processor, however, the program behaves non-deterministically as expected, for both the original and record versions.

Figure 18 summarises the results from running the replay version of the `tsp` program. The trace tree used is from one of the executions with 32 processors for the record version, shown in Figure 17. The first row ('rec') shows the measurements for the one record run that is used for the replay. This is one of the three runs used to obtain the results for the 32 processor run of the record version presented in Figure 17. The other rows in Figure 18 show the results for running the replay program using 1, 16, and 32 processors. Again each configuration is run three times, and the averages, with standard deviations, are shown. As can be seen, the replay version expands exactly the same number of nodes as the version it is recorded from, thus showing that the replay version correctly reproduces the behaviour of the recorded program. The number of reductions does vary slightly, apparently because of the way built-ins are measured in PIM/M: if they are suspended, they will be counted more than once. As the time and number of suspensions can still vary in the replay version, some differences are expected. Indeed, the differences are much smaller than the variations seen in different runs of the original and record version of the program, where the reduction differences are due to truly non-deterministic behaviour of the program.

No timings are shown for the replay version because, in this case, speed is not critical: we expect execution to be replayed only in conjunction with debugging tools that allow the execution to be examined, which will slow down the execution considerably. However, the actual time to execute a replay program without tracing should be similar to the original execution times. We were unable to verify this as we cannot exclude the time for reading in the trace tree, which is quite slow on the PIMs.

[4] This is because all solver processes were given equal priority, and so the number of nodes expanded increases with the number of processes. In the original Parlog program, lower priority was given to solvers performing speculative work, and so a speedup was achieved.

In summary, our experiments with the Instant Replay scheme on PIM/M verify that the scheme does behave as we expect in a parallel environment. In addition, recording an execution introduces very little perturbation.

6 Conclusions and Further Work

The Instant Replay approach to debugging seems to be a feasible way of providing repeatable execution for KLIC programs at a low cost. We have demonstrated that the replay version of the program is indeed able to repeat the execution of the record version, for both sequential KLIC and for parallel KL1 running on the PIMs. Repeatable execution solves a major problem with debugging concurrent logic programs that are actually executed in parallel, and is useful for most, if not all, debugging approaches for concurrent logic programming languages. It is also useful for performance monitoring and performance debugging. We believe that the method we used is very general, and can be easily applied to other concurrent logic programming languages.

It is important to stress that providing the ability to repeat executions does not solve the problem of debugging concurrent logic programs: debuggers or other methods which can be used to help the user examine and understand the execution are still needed.

The Instant Replay system that we have implemented is currently being integrated into the KLIC environment so that the monitoring process is invisible to the user. This requires modifying the KLIC debugging tools so that the user sees the original program being debugged, instead of the converted version, which is much less humanly readable.

Future work includes examining more non-deterministic KLIC programs to see how the scheme behaves in more examples. It may also be interesting to investigate implementing the method at a lower level than a source-to-source transformation, as this would be even more efficient, especially if very much monitoring needs to be done. There is also ample scope for improving the algorithms for determinism analysis. For example, we currently use a bottom-up algorithm to detect total determinism, which has the disadvantage that it fails to exploit information about how each predicate is actually called, and therefore loses precision.

7 Acknowledgements

We gratefully acknowledge the support of ICOT for this work, and for the help they gave us during our visits. We would also like to thank Vítor Santos Costa, John Gallagher, Manuel Hermenegildo, Manuel Carro, Enrico Pontelli for helpful discussions on abstract interpretation, and to John Lloyd for discussions on declarative and algorithmic debugging. Jimmy Lee helped with clarifying some of the definitions. Hiroyoshi Hatazawa was most helpful in doing most of the work in converting our system from KLIC to KL1 for the PIMs, and helping us to use the PIMs at ICOT. We thank Takashi Chikayama for many general discussions on our system, and for the development of the postmortem processing for KLIC which allowed our scheme to deal with error conditions.

References

[1] M. Brayshaw. An Architecture for Visualizing the Execution of Parallel Logic Programs. In *International Joint Conference on Artificial Intelligence*, 1991.

[2] T. Chikayama, T. Fujise, and D. Sekita. A Portable and Efficient Implementation of KL1. In T. Chikayama and E. Tick, editors, *Workshop on Parallel Logic Programming and its Programming Environments*, 1994.

[3] T. Conlon and S. Gregory. Debugging Tools for Concurrent Logic Programming. *The Computer Journal 35*, 2, pages 157–169, 1992.

[4] M. Ducassé. Abstract Views of Prolog Executions in Opium. In V. Saraswat and K. Ueda, editors, *Logic Programming: Proceedings of 1991 International Symposium*, pages 18–32. The MIT Press, 1991.

[5] S. Gregory. Experiments with Speculative Parallelism in Parlog. In D. Miller, editor, *Logic Programming: Proceedings of the 1993 International Symposium*, pages 370–388. The MIT Press, 1993.

[6] M. M. Huntbach. Algorithmic Parlog Debugging. In *Proceedings of the Fourth Symposium on Logic Programming*. Computer Society Press of the IEEE, Sept. 1987.

[7] R. A. Kowalski. *Logic for Problem Solving*. Artificial Intelligence Series. North Holland, 1979.

[8] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers C-36*, 4, pages 471–481, 1987.

[9] V. Santos Costa. *Compile-time Analysis for the Parallel Execution of Logic Programs in Andorra-I*. PhD thesis, University of Bristol, 1993.

[10] E. Shapiro. *Algorithmic Program Debugging*. The MIT Press, 1983.

[11] A. Takeuchi. *Parallel Logic Programming*. John Wiley & Sons, Inc., 1992.

[12] J. Tatemura, H. Koike, and H. Tanaka. HyperDEBU: A Multiwindow Debugger for Parallel Logic Programs. Technical report, University of Tokyo, 1992.

[13] E. Tick. *Parallel Logic Programming*. The MIT Press, 1991.

[14] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal 33*, 6, pages 494–500, 1990.

[15] R. Yang. CHUKL: Constraint Handling Under KL1. In T. Chikayama and E. Tick, editors, *Workshop on Parallel Logic Programming and its Programming Environments*, 1994.