Schedule Analysis of Concurrent Logic Programs

Andy King and Paul Soper
Department of Electronics and Computer Science,
University of Southampton, Southampton, S09 5NH, UK
amk@ecs.soton.ac.uk, pjs@ecs.soton.ac.uk

Abstract

A compilation technique is proposed for concurrent logic programs called schedule analysis. Schedule analysis deduces at compile-time a partial schedule for the processes of a program by partitioning the atoms of each clause into threads. Threads are totally ordered sets of atoms whose relative ordering is determined by a scheduler. Threads reduce scheduler activity and permit a wealth of traditional Prolog optimisations to be applied to the program. A framework for schedule analysis is proposed and this defines a procedure for creating threads. A safety result is presented stating the conditions under which the work of the scheduler can be reduced from ordering processes to ordering threads. Schedule analysis has been integrated into a compiler and implementation has suggested that it can play a central rôle in compilation. Optimisations which follow from schedule analysis include a reduction in scheduling, the removal of synchronisation checks, the simplification of unification, decreased garbage collection and a reduction in argument copying.

1 Introduction

Concurrent logic programming brings a new dimension of expressiveness to logic programming enabling a host of useful protocols and paradigms to be modeled. This flexibility has a cost, however, because it is the control strategy of Prolog which has bought efficient implementation. The depth-first search of Prolog, for instance, brings with it a stack to support local variables and continuations. On the other hand, concurrent logic programs require scheduling which introduces the extra overheads of enqueuing and dequeuing processes. Furthermore, without continuations argument copying is increased and without local variables garbage collection becomes more frequent. This is the penalty of substituting data-flow for control-flow. Schedule analysis shows how to selectively replace data-flow with control-flow and thereby reduce these overheads.

Schedule analysis is concerned with deducing at compile-time a partial schedule of processes, or equivalently the body atoms of a clause, which is consistent with the program behaviour. Program termination characteristics are affected if an atom which binds a shared variable is ordered after an atom that matches on that variable. In order to avoid this, an ordering of the atoms is determined which does not contradict any data-dependence of the program. In general the processes cannot be totally ordered and thus the analysis leads to a division into threads of totally ordered processes. In this way the work required of the run-time scheduler is reduced to ordering threads.

Scheduling threads instead of processes avoids the creation of unnecessary suspensions during evaluation. This is useful because the overhead incurred by each suspension is significant. The overhead is not merely in the extra enqueuing and dequeuing of the process, since upon resumption of the process, the guards of the associated predicate usually have to be retried. In addition to avoiding the creation of unnecessary suspensions, schedule analysis permits several useful optimisations to be applied within a thread. The optimisations all depend on the existence of a total ordering of atoms within a thread and follow from applying mode analysis, type analysis and reference analysis to the threads [8]. Mode analysis and type analysis can be used to identify: instances of unification which can be simplified; repeated synchronisation instructions which can be removed; and redundant checks which can be removed when producers are ordered before consumers in the same thread. Reference analysis can be used to identify: variables which can be accessed without dereferencing; variables for which initialisation and unification can be simplified; and local variables which can be allocated to a stack.

Schedule analysis exchanges parallelism for reduced overheads by inferring orderings for atoms. In this sense schedule analysis addresses some of the fundamental issues involved in implementing a concurrent logic program on a uniprocessor. For a multi-processor, however, there is a danger of introducing too much control-flow and therefore limiting parallelism; a balance between control-flow and data-flow needs to be struck. One way to get an efficient and balanced untilisation of a multi-processor is to partition a program into grains [4]. A grain is a set of processes, to be executed on a single processor, when it is less efficient to evaluate them in parallel. For a concurrent logic program, a consequence of data-flow is that there is no explicit ordering between the processes within a grain. Thus the processes within a grain have to be scheduled. Thus, by introducing control-flow within the scope of a grain, the benefits of reduced overheads can be obtained without compromising the parallelism. The principle is therefore to turn excess or fine-grained parallelism to good use by exchanging the parallelism for reduced overheads. King and Soper [7] describe in detail how to systematically identify fine-grained parallelism and appropriately introduce control-flow to remove the ineffective parallelism.

Section 2 describes the notation and preliminary definitions which will be used throughout. Section 3 develops a framework for schedule analysis in which the data-dependencies between the atoms of a clause are characterised in a way which is independent of the query. This enables pairs of atoms to be identified which must be allocated to different threads. Theorem 3.1, a safety result, states the conditions under which atoms can be partitioned into threads and ordered within a thread whilst preserving the behaviour of the program. This framework, however, is not tractable and section 4 outlines a practical procedure for constructing threads and gives some preliminary results. Sections 5 and 6 present related work and the concluding discussion.

2 Notation and preliminaries

To introduce schedule analysis some notation and preliminary definitions are required. Let *Atom* denote the set of atoms for a program *Prog*, with typical member $a \in Atom$. Additionally let *Goal* represent the set of goals, that is $Goal = \wp(Atom)$, with typical member $g \in Goal$. For generality, let *Clause* denote the set of clauses of the form a `<-` $a_{ask,1}$, ..., $a_{ask,i}$: $a_{tell,1}$, ..., $a_{tell,j}$ `|` $a_{body,1}, \ldots, a_{body,k}$. The ask, tell and body atoms of a clause c are represented by the ordered sets $ask_c = \{a_{ask,1}, \ldots, a_{ask,i}\}$, $tell_c = \{a_{tell,1}, \ldots, a_{tell,j}\}$ and $body_c = \{a_{body,1}, \ldots, a_{body,k}\}$. If there are no tell atoms the `|` connective is omitted from a clause, whereas if there are neither any ask atoms nor any tell atoms both the : and `|` connectives are omitted from a clause. A concurrent logic program *Prog* is a finite set of clauses. Let *Subs* denote the set of substitutions for *Prog* with typical member θ. Additionally let the set of states of a transition relation for *Prog* be denoted by *State*, where $State = Goal \times Subs$, and *State* has typical member s.

Definition 1 (match, try and the transition relation) *The mappings* $match : Atom \times Atom \to \{fail, susp\} \cup Subs$, $try : Atom \times Clause \to \{fail, susp\} \cup Subs$ *and the transition relation* $\vdash$ *on State are defined by:*

$$match(a', c) = \begin{cases} \theta' & \theta \in mgu(a', a), \\ & \langle ask_c, \theta \rangle \vdash^* \langle \emptyset, \theta' \rangle, \\ & \theta' \upharpoonright vars(a') = \epsilon. \\ susp & \theta \in mgu(a', a), \\ & \langle ask_c, \theta \rangle \vdash^* \langle \emptyset, \theta' \rangle, \\ & \theta' \upharpoonright vars(a') \neq \epsilon. \\ fail & otherwise. \end{cases}$$

$$try(a', c) = \begin{cases} \theta' & \theta \in match(a', c), \\ & \langle tell_c, \theta \rangle \vdash^* \langle \emptyset, \theta' \rangle. \\ susp & susp = match(a', c). \\ fail & otherwise. \end{cases}$$

where $\vdash^*$ *is the reflexive and transitive closure of* $\vdash$, $\vdash$ *is written infix,* $\theta \upharpoonright V$ *denotes the restriction of a substitution* θ *to a set of variables* V, $vars(a)$ *denote the variables of* a, *and* $mgu(a, b)$ *is the set of most general unifiers for* a *and* b.

The transition relation for Prog is described piece-wise in terms of transition relations for the clauses of Prog. The transition relation for a clause

c is described by: if $try(a_m\theta, c') = \theta'$, c' is renamed apart from c and $body_{c'} = \{a'_1, \ldots, a'_k\}$ then $\langle \{a_1, \ldots, a_n\}, \theta\rangle \vdash^{c,c'} \langle \{a_1, \ldots, a_{m-1}, a'_1, \ldots, a'_k, a_{m+1}, \ldots, a_n\}, \theta \circ \theta'\rangle$.

The transition relation for Prog is described by: $\langle g, \theta\rangle \vdash \langle g', \theta'\rangle$ if there exists a clause c of Prog such that $\langle g, \theta\rangle \vdash^{c,c'} \langle g', \theta'\rangle$.

The transition relation for *Prog* engenders the notion of a proof. Thus let *Proof* denote the set of proofs for *Prog* with typical member p. A proof is a sequence $p = s_i, s_{i+1}, s_{i+2}, \ldots$ such that $s_i \vdash s_{i+1}$, $s_{i+1} \vdash s_{i+2}, \ldots$ and can be either finite or infinite. (The term proof has been extended to include infinite sequences for notational convenience.) Operationally a proof corresponds to a computation. A computation $p = s_i, s_{i+1}, \ldots, s_j$ is suspended if there does not exist $s \in State$ such that $s_j \vdash s$. Additionally p is resumed by ϑ if p is suspended with $s_j = \langle g, \theta\rangle$ and there exists $s \in State$ such that $\langle g, \theta \circ \vartheta\rangle \vdash s$. Alternatively if $g = \emptyset$ then p is terminated.

3 A framework for schedule analysis

The abstract framework formalises the notion of a data-dependence between two atoms of a goal and explains how to translate data-dependencies into threads.

3.1 Data-dependencies among the atoms of a goal

The existence of a data-dependency between two atoms is inferred by studying the order in which the atoms are resolved within a computation. A partial mapping resolve is used to indicate which resolution steps of a computation are responsible for solving a particular atom of the goal. The mapping resolve is defined in terms of the partial mapping solve.

Definition 2 (solve) *The partial mapping solve : $Proof \times Atom \times \mathbb{N} \to Goal$ is defined by:*

$$solve(p, a, j) = \begin{cases} \{a\} & i = j, a \in g_i. \\ \{a' \mid a' \in solve(p, a, j-1), a' \neq a_m\} \cup & \\ \{a'_1, \ldots, a'_k\} & i < j, a_m \in solve(p, a, j-1). \\ solve(p, a, j-1) & i < j, a_m \notin solve(p, a, j-1). \end{cases}$$

where $p = s_i, s_{i+1}, \ldots, s_{j-1}, s_j, \ldots, s_j = \langle g_j, \theta_j\rangle$, $g_{j-1} = \{a_1, \ldots, a_m, \ldots, a_n\}$ and $g_j = \{a_1, \ldots, a_{m-1}, a'_1, \ldots, a'_k, a_{m+1}, \ldots, a_n\}$.

Definition 3 (resolve) *The partial mapping resolve : $Proof \times Atom \to \wp(\mathbb{N})$ is defined by:*

$$resolve(p, a) = \{j \mid a' \in solve(p, a, j), a' \notin solve(p, a, j+1)\}.$$

```
canDepend(Y) <- X = 1, canConsume(X, Y).

canConsume(X, Y) <- X = Y.
canConsume(X, Y) <- Y is X + 1.
```

Figure 1: The canDepend/1 and canConsume/2 predicates.

Intuitively a data-dependency exists from one atom to another, if the computation for the second atom can never entirely precede the computation for the first atom. The resolve mapping is useful because it indicates how atoms are scheduled thus enabling a data-dependence to be identified. Definition 5 formalises the idea of a data-dependence between two atoms which, in turn, is defined in terms of whether an atom finishes or persists.

Definition 4 (finish and persist) *If $s_i = \langle g_i, \theta_i \rangle$, $p = s_i, s_{i+1}, \ldots$ and $a \in g_i$ then*

- *a finishes in p at l if there exists $l \geq i$ such that solve$(p, a, l) \neq \emptyset$ and solve$(p, a, l+1) = \emptyset$,*
- *a persists in p if solve$(p, a, l) \neq \emptyset$ for all $l \geq i$.*

Definition 5 (data-dependence) *Suppose $s_i = \langle g_i, \theta_i \rangle$ and $a_1, \ldots, a_j, a' \in g_i$ with $a_1 \neq a'$, ..., $a_j \neq a'$. There exists a data-dependence from a_1 or ...or a_j to a' for $p = s_i, \ldots, s_k$ if j is the least j such that*

- *for all $p' = s_i, \ldots, s_k, s_{k+1}, \ldots$ if a' finishes in p' at l then there exists a_m with $1 \leq m \leq j$ such that $n \in$ resolve(p', a_m) and $k < n < l$.*
- *for all $p' = s_i, \ldots, s_k, s_{k+1}, \ldots$ if a' persists in p' then there exists a_m with $1 \leq m \leq j$ such that $n \in$ resolve(p', a_m) and $k < n$.*
- *there exists $p' = s_i, \ldots, s_k, s_{k+1}, \ldots$ such that either a' finishes in p' at l or a' persists in p'.*

Note that definition 5 considers all p such that $p' = s_i, \ldots, s_k, s_{k+1}, \ldots$ possibly including p which are infinite, suspended and terminated. The rôle of $p = s_i, \ldots, s_k$ in definition 5 is technical and chiefly deals with non-determinism. Its inclusion in definition 5 is necessary because, in general, the presence of a data-dependence can be decided by a non-deterministic choice. Example 1 uses the contrived but illuminating canDepend/1 and canConsume/2 predicates listed in figure 1 to demonstrate how non-determinism can determine the existence of a data-dependence. The example illustrates the significance of $p = s_i, \ldots, s_k$ in definition 5.

Example 1 *The non-determinism in the* canConsume/2 *predicate of figure 1 decides whether or not a data-dependence exists from the* X = 1 *atom to the* canConsume(X, Y) *atom of the* canDepend/1 *clause. The data-dependence does not exist for p_1 but does exist for p_2 where $p_1 = s_1, s_2, s_3, s_4, s_5$ and $p_2 = s_1, s_2, s'_3, s'_4, s'_5$ are given below. The initial states of p_1 and p_2, s_1 and s_2, coincide. Different clauses of* canConsume/2, *however, derive s_3 and s'_3 so that s_3 and s'_3 and the proceeding states differ.*

$s_1 = \langle\ \{\mathit{canDepend(Y)}\}, \epsilon\ \rangle \vdash$
$s_2 = \langle\ \{X = 1, \mathit{canConsume}(X,\ Y_1)\}, \{Y_1 \mapsto Y\}\ \rangle \vdash$
$s_3 = \langle\ \{X = 1, X = Y_1\}, \{Y_1 \mapsto Y\}\ \rangle \vdash$
$s_4 = \langle\ \{X = 1\}, \{Y_1 \mapsto Y, X \mapsto Y_1\}\ \rangle \vdash$
$s_5 = \langle\ \emptyset, \{Y \mapsto 1, X \mapsto 1, Y_1 \mapsto 1\}\ \rangle.$

$s'_3 = \langle\ \{X = 1, Y_1 \text{ is } X + 1\}, \{Y_1 \mapsto Y\}\ \rangle \vdash$
$s'_4 = \langle\ \{Y_1 \text{ is } X + 1\}, \{Y_1 \mapsto Y, X \mapsto 1\}\ \rangle \vdash$
$s'_5 = \langle\ \emptyset, \{Y \mapsto 2, X \mapsto 1, Y_1 \mapsto 2\}\ \rangle.$

For p_1, no data-dependencies can exist from X = 1 *to* canConsume(X, Y) *since resolve(s_2, s_3, s_4, s_5,* canConsume(X, Y)*) = $\{2, 3\}$ and resolve(s_2, s_3, s_4, s_5,* X = 1*) = $\{4\}$. For p_2, however, $3 \in$ resolve(s_2, s'_3, s'_4, s'_5,* X = 1*) and $4 \in$ resolve(s_2, s'_3, s'_4, s'_5,* canConsume(X, Y)*). Putting $p = s_2, s'_3$ illustrates the rationale behind $p = s_i, \ldots, s_k$ in definition 5. p and hence s'_3 are fixed therefore the choice of clause for* canConsume(X, Y) *is predefined. Therefore, a data-dependence always occurs in $s_{k+1}, s_{k+2}, \ldots$ allowing the data-dependence from* X = 1 *to* canConsume(X, Y) *to be identified and captured.*

3.2 Data-dependencies among the atoms of a clause

Schedule analysis focuses on organising the body atoms of a clause c, $body_c$, into threads. Therefore the notion of a data-dependence between the body atoms of a clause is introduced.

Definition 6 (data-dependence relation) *The data-dependence relation $\delta_{c,p}$ for a clause c and a computation $p = s_i, \ldots, s_k$ is a relation on $body_c$ is defined by: if there exists $s_{i-1} \in State$ such that $s_{i-1} \vdash^{c,c'} s_i$, and $body_c = \{a_1, \ldots, a_k\}$, $body_{c'} = \{a'_1, \ldots, a'_k\}$, and there exists a data-dependence from a'_{l_1} or $\ldots$ or a'_{l_j} to $a'_{l_{j+1}}$ for p then $\langle a_{l_1}, a_{l_{j+1}}\rangle, \ldots, \langle a_{l_j}, a_{l_{j+1}}\rangle \in \delta_{c,p}$.*

Example 2 *Let fib denote the recursive clause of the the* fib/2 *predicate presented in figure 2. Additionally let a_1 =* N1 is N - 1, *a_2 =* N2 is N - 2, *a_3 =* fib(N1, F1), *a_4 =* fib(N2, F2) *and a_5 =* F is F1 + F2 *so that $body_{fib} = \{a_1, \ldots, a_5\}$. Since fib is deterministic consider $p = s_1$ where $s_1 = \langle\{a_1, \ldots, a_5\}, \{N \mapsto 1\}\rangle$. The data-dependence relation $\delta_{fib,p}$ is presented as a directed graph in figure 3.*

```
fib(N, F) <- N =< 1 : F = 1.
fib(N, F) <- N > 1 : N1 is N - 1, N2 is N - 2,
    fib(N1, F1), fib(N2, F2), F is F1 + F2.
```

Figure 2: The fib/2 predicate.

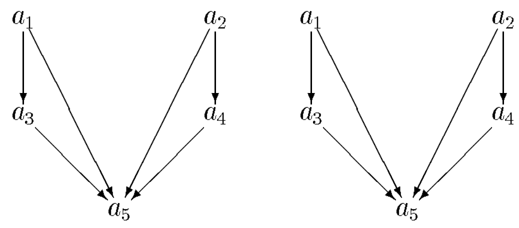

Figure 3: $\delta_{fib,p}$ and δ_{fib}.

Although the data-dependence relation $\delta_{fib,p}$ is acyclic, a data-dependence relation can contain cycles due to the possibility of coroutining, each cycle corresponding to a set of coroutining atoms. It is important that schedule analysis identifies coroutining since coroutining atoms need to be allocated to different threads. Once placed in separate threads the data-dependencies can be resolved at run-time with a scheduler.

Usually compilers do not have the benefit of a knowledge of the computation p and hence it is necessary to derive a data-dependence relation which is independent of p. In the terminology of abstract interpretation, $\delta_{c,p}$ is collected for each possible p, to construct a relation δ_c on $body_c$ which summarises the data-dependencies of $body_c$ in a way which independent of p.

Definition 7 (collecting data-dependence relation) *A relation δ_c on $body_c$ is a collecting data-dependence relation for a clause c if: $\delta_{c,p} \subseteq \delta_c$ for all p.*

If each data-dependence relation is acyclic, coroutining cannot occur. Nevertheless, cycles can still appear in a collecting data-dependence relation. This is symptomatic of atoms whose scheduling order depends on the initial computation, for instance, the query. In addition to the coroutining atoms, these atoms need to be placed in different threads so that they can be ordered at run-time by the scheduler. Example 3 illustrates how cyclic data-dependencies can be introduced into the least collecting data-dependence relation by atoms whose relative scheduling ordering depends on the query.

Example 3 *Figure 4 details the pathological abc/2, a/3 and b/3 predicates. These predicates are contrived so that for some queries AA is A + 5 is*

```
abc(X, C) <- C is A + B, AA is A + 5, BB is B + 6,
    a(X, BB, A), b(X, AA, B).

a(X, BB, A) <- 0 < X : A = BB.
a(X, _, A) <- 0 >= X : A = 3.

b(X, _, B) <- 0 < X : B = 4.
b(X, AA, B) <- 0 >= X : B = AA.
```

Figure 4: The abc/2, a/3 and b/3 predicates.

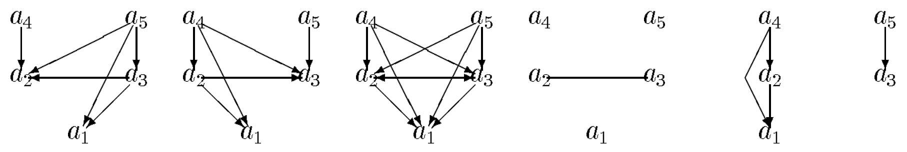

Figure 5: δ_{abc,c^+}, δ_{abc,c^-}, δ_{abc}, σ_{abc} and τ_{abc}.

scheduled before BB is B + 6 *whereas, for others, the ordering is reversed. Specifically, the order of scheduling is predicated on* X > 0. *Let abc denote the clause the* abc/3 *predicate and* a_1 = C is A + B, a_2 = AA is A + 5, a_3 = BB is B + 6, a_4 = a(X, BB, A), a_5 = b(X, AA, B) *so that* $body_{abc} = \{a_1, \ldots, a_5\}$. *Further let* c^+, c^- *and* $c^\perp$ *respectively denote* $c = s_i, \ldots, s_j$ *for which* s_i *satisfies* X > 0, s_i *satisfies* 0 >= X, *and for which* s_i *neither satisfies* X > 0 *nor* 0 >= X *because* X *is unbound. Thus* $\delta_{abc} = \delta_{abc,c^+} \cup \delta_{abc,c^-} \cup \delta_{abc,c^\perp}$. *Since* $c^\perp$ *immediately leads to a suspension,* $\delta_{abc,c^\perp} = \emptyset$, *so that* $\delta_{abc} = \delta_{abc,c^+} \cup \delta_{abc,c^-}$. *Figure 5 presents the relations* δ_{abc,c^+}, δ_{abc,c^-} *and* δ_{abc} *as directed graphs. Note that* δ_{abc} *is cyclic even though* δ_{abc,c^+} *and* δ_{abc,c^-} *are both acyclic.*

3.3 Data-dependencies among the threads of a clause

A collecting data-dependence relation is used to partition the atoms of a clause into threads. Specifically threads are formed by identifying pairs of atoms which must be allocated to different threads. There are just four ways in which data-dependencies can occur between a pair of atoms. These four ways are listed and categorised in figure 6. The data-dependencies in each category have different implications for the scheduling. By identifying which category requires run-time scheduling, a prescription for generating threads is derived.

For category one, either a data-dependence always exists from a to a' or sometimes exists from a to a'. Thus a can be ordered before a' within

Category	*Characteristic*	*Order*
1	$\langle a, a' \rangle \in \delta_c$ and $\langle a', a \rangle \notin \delta_c$	a precedes a'.
2	$\langle a', a \rangle \in \delta_c$ and $\langle a, a' \rangle \notin \delta_c$	a' precedes a.
3	$\langle a, a' \rangle \notin \delta_c$ and $\langle a', a \rangle \notin \delta_c$	neither a precedes a' nor a' precedes a.
4	$\langle a, a' \rangle \in \delta_c$ and $\langle a', a \rangle \in \delta_c$	either a precedes a' or a' precedes a, or a and a' coroutine.

Figure 6: Categorising atom pairs.

the same thread at compile-time. Category two is the symmetric variant of category one. For category three, a data-dependence neither exists from a to a' nor from a' to a. Thus a and a' can be ordered at compile-time in any manner! Category four either locates coroutining activity in which data-dependencies exist both from a to a' and from a' to a; or identifies computations for which a data-dependence exists from a to a' in one computation and from a' to a in another. In either case, the atoms a and a' must be assigned to different threads and the ordering of a and a' resolved at run-time. Of these four categories only category four corresponds to pairs of atoms which require run-time scheduling. A relation σ_c on $body_c$ is introduced, called the separation relation, which is used to isolate pairs of atoms which have to be allocated to different threads.

Definition 8 (separation relation) *A relation σ_c on $body_c$ is a separation relation for a clause c if: $\langle a, a' \rangle \in \sigma_c$ if $\langle a, a' \rangle \in \delta_c$ and $\langle a', a \rangle \in \delta_c$.*

A separation relation σ_c is symmetric, that is, if $\langle a, a' \rangle \in \sigma_c$ then $\langle a', a \rangle \in \sigma_c$, and therefore can be represented pictorially as an undirected graph. σ_c is used to partition $body_c$ into sets of atoms which, when ordered, become threads.

Definition 9 (partition) *A set $part_c$ is a partition of a clause c if:*

- $part_c = \{part_{c,1}, \ldots, part_{c,t}\}$,
- $part_{c,1} \cup \ldots \cup part_{c,t} = body_c$,
- $part_{c,i} \cap part_{c,j} = \emptyset$ *for all* $i \neq j$,
- *if* $\langle a, a' \rangle \in \sigma_c$ *then* $a \in part_{c,i}$, $a' \in part_{c,j}$ *with* $i \neq j$.

To turn a partition $part_c$ into threads, each $part_{c,i}$ is ordered so as not to contradict a data-dependence in δ_c.

```
incr(X, Y) <- Y is X + 1.

incrs(Y1, Y2) <- X1 = 1, X2 = 2, incr(X1, Y1), incr(X2, Y2).
```

Figure 7: The incr/2 and incrs/2 predicates.

Definition 10 (thread orderings) *A set o_c is a set of thread orderings for a clause c if:*

- $o_c = \{o_{c,1}, \ldots, o_{c,t}\}$,
- $o_{c,i}$ *is a relation on* $part_{c,i}$,
- $o_{c,i} = \{\langle a_{i_k}, a_{i_l}\rangle \mid 1 \le k < l \le m\}$ *if* $part_{c,i} = \{a_{i_1}, \ldots, a_{i_m}\}$,
- *if* $\langle a, a'\rangle \in o_{c,i}$ *then* $\langle a', a\rangle \notin \delta_c^+$.

The transitive closure of a relation δ_c is denoted by δ_c^+.

Example 4 *Figure 3 diagrams δ_{fib}. Since δ_{fib} is acyclic, $\sigma_{fib} = \emptyset$, and therefore $part_{fib} = \{body_{fib}\}$ is a partition of c. In the notation of example 2, take $o_{fib,1} = \{\langle a_i, a_j\rangle \mid 1 \le i < j \le 5\}$. The clause abc is more illuminating since δ_{abc} is cyclic and therefore $\sigma_{abc} \ne \emptyset$. Adopting the atom labeling used in example 3, $\sigma_{abc} = \{\langle a_2, a_3\rangle, \langle a_3, a_2\rangle\}$. σ_{abc} is represented pictorially as an undirected graph in figure 5. Therefore $part_{abc} = \{part_{abc,1}, part_{abc,2}\}$ is a partition of abc where $part_{abc,1} = \{a_4, a_2, a_1\}$ and $part_{abc,2} = \{a_5, a_3\}$. In turn, the partition leads of the threads $o_{abc,1} = \{\langle a_4, a_2\rangle, \langle a_2, a_1\rangle, \langle a_4, a_1\rangle\}$ and $o_{abc,2} = \{\langle a_5, a_3\rangle\}$.*

Although threads are constructed so as to avoid contradicting any data-dependencies of the program, threads can be formed which compromise termination. Example 5 uses the incr/2 and incrs/2 predicates listed in figure 7 to illustrate circumstances in which the threads produced by schedule analysis inadvertently introduce deadlocking behaviour.

Example 5 *The incr/2 and incrs/2 predicates are presented in figure 7. Let incrs denote the clause of the incrs/2 predicate and let a_1 = X1 = 1, a_2 = X2 = 2, a_3 = incr(X1, Y1), a_4 = incr(X2, Y2) so that $body_{incrs} = \{a_1, a_2, a_3, a_4\}$. δ_{incrs} is acyclic and is diagrammed in figure 8. Since δ_{incrs} is acyclic a single thread can be constructed but, for the sake of a counterexample, divide $body_{incrs}$ according to $part_{incrs} = \{part_{incrs,1}, part_{incrs,2}\}$ where $part_{incrs,1} = \{a_1, a_4\}$ and $part_{incrs,2} = \{a_2, a_3\}$. In addition, choose the threads $o_{incrs,1} = \{\langle a_4, a_1\rangle\}$ and $o_{incrs,2} = \{\langle a_3, a_2\rangle\}$. $o_{incrs,1}$ and $o_{incrs,2}$ are also presented in figure 8. Note, however, that $\delta_{incrs} \cup o_{incrs,1} \cup o_{incrs,2}$ is cyclic. In operational terms, deadlock occurs, since a_4 of $o_{incrs,1}$ suspends waiting for a_2 to bind X2 and a_3 of $o_{incrs,2}$ suspends waiting for a_1 to bind X1.*

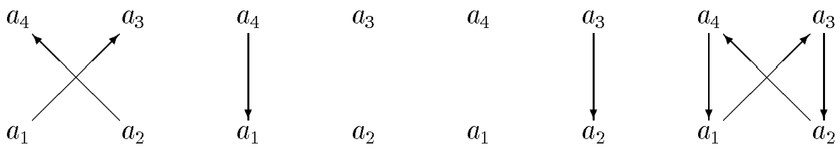

Figure 8: δ_{incrs}, $o_{incrs,1}$, $o_{incrs,2}$ and $\delta_{incrs} \cup o_{incrs,1} \cup o_{incrs,2}$.

Deadlock can occur because the constraints on the scheduling of the threads can extend beyond the constraints on the scheduling of the clause. The scheduling constraints for the threads, however, augment δ_c with the orderings imposed by the threads themselves. The cumulative ordering effect of the threads is given by $o_{c,1} \cup \ldots \cup o_{c,t}$ which is dubbed τ_c.

Definition 11 *τ_c is a relation on $body_c$ defined by: $\tau_c = o_{c,1} \cup \ldots \cup o_{c,t}$.*

The scheduling constraints on the threads therefore extend from δ_c to $\tau_c \cup \delta_c$. Example 5 illustrates the disastrous effect that extra cycles in $\tau_c \cup \delta_c$ can have on termination. Note, however, that if all the cycles which occur in $\tau_c \cup \delta_c$ also occur in δ_c, then termination is not compromised. This follows because the atoms which form the cycles of $\tau_c \cup \delta_c$ would occur in the cycles of δ_c and would therefore be split across different threads by schedule analysis. Theorem 3.1 confirms this observation stating the precise conditions under which the work of a scheduler can be safely reduced from scheduling processes to scheduling threads. The theorem thus provides a way for checking the integrity of the threads generated by schedule analysis. A reliable procedure for forming threads thus follows, since erroneous threads can be identified and be removed.

To introduce the safety theorem, however, the notion of an interleave is required to express a scheduling order of the threads. A scheduler can transfer control from one thread to another by suspending and resuming the evaluation of a thread. The scheduler therefore induces an ordering on $body_c$ which extends beyond τ_c. The concept of an interleave is introduced in definition 12 to capture this ordering and clarify the way in which the evaluation of the threads can be interleaved.

Definition 12 *An interleave of τ_c is a relation ι_c on $body_c$ defined by:*

- *if $\langle a, a' \rangle \in \iota_c$ then $\langle a', a \rangle \notin \tau_c$,*
- *if $a, a' \in body_c$ and $a \neq a'$ then either $\langle a, a' \rangle \in \iota_c$ or $\langle a', a \rangle \in \iota_c$.*

Theorem 3.1 (safety theorem [9]) *If $\tau_c \cup \delta_c^+$ has no more cycles than δ_c^+ then there exists an interleave ι_c of τ_c for arbitrary c.*

4 A procedure for schedule analysis

The framework can be regarded a compilation scheme in which the input is a collecting data-dependence relation and the output is the threads of a clause. Thus, once a collecting data-dependence relation is found, schedule analysis can be integrated into a compiler. King and Soper [9] explain how data-dependencies can be inferred with existing forms of analysis, specifically producer and consumer analysis [6, 1] or the mode algorithm of Ueda and Morita [13]; detail how a collecting data-dependence relation can be constructed; and show how the framework defines a constructive procedure, an algorithm, for generating threads.

Schedule analysis has been integrated into a compiler for a dialect of Flat Parlog. The final version of the schedule analysis module, which excludes the producer and consumer analysis, equates to about 350 lines of code. The bulk of the module equates to code for calculating the transitive closure of a relation, counting the number of cycles in a relation, and computing a partition. The transitive closure and cycle count is computed by variants of the backtracking algorithm proposed by Tiernan [12]. The partitioning is accomplished by an approximation algorithm, specifically the largest-first sequential colouring algorithm [11]. Although computing an optimal partition is equivalent to calculating $\chi(\sigma_c)$ which, in turn, is NP-complete [3], the approximation algorithm was found to frequently minimise the number of threads.

Some of the optimisations which follow from schedule analysis are best illustrated at the level of the intermediate code. The intermediate code used in the compiler is similar to Kernel Parlog [2]. Kernel Parlog is a useful basis for the intermediate language because it includes control primitives like Data/1[1] and supports the sequential and the parallel conjunction. The primitive Data(X) induces suspension if X is unbound. The parallel and the sequential conjunction, respectively denoted , and &, provide a way to order the evaluation of atoms within a clause and thereby express threads. The usefulness of Kernel Parlog extends beyond that of a representation language since Gregory (1987) proposed a suite of optimisations that can be used in connection with sequential conjunction and therefore apply to threads. The optimisations are significant because they can be used as a measure of the effectiveness of schedule analysis.

Figure 9 lists two versions of the intermediate code for fib/2, generated without and with schedule analysis. To put the optimisations into perspective, the instruction count for the Data/1 and Unify/2 primitives drops from 529 and 353 to just 1 and 177 when the tenth Fibonacci number is computed. The other primitives Less/2, Less_Equal/2, Minus/3 and Plus/3 are

[1] To be faithful to the intermediate code used in the compiler, primitives are denoted by Data/1, Less/2, Less_Equal/2, Plus/3 and Unify/2 rather than using the notation DATA/1, LESS/2, LESSEQ/2, PLUS/3 and =/2 introduced by Gregory (1987). The extra primitive Minus/3 required for fib/2 has the obvious interpretation.

```
fib(_1, _2) <- Data(_1) & Less_Equal(_1, 1) : Unify(_2, 1).
fib(_1, _2) <-
    Data(_1) & Less(1, _1) :
    Data(_1) & Minus(_1, 1, _4) & Unify(_4, N1),
    Data(_1) & Minus(_1, 2, _5) & Unify(_5, N2),
    fib(N1, F1), fib(N2, F2),
    Data(F1) & Data(F2) & Plus(F1, F2, _6) & Unify(_6, _2).

fib(_1, _2) <- true : Data(_1) & $fib(_1, _2).

$fib(_1, _2) <- Less_Equal(_1, 1) : Unify(_2, 1).
$fib(_1, _2) <-
    Less(1, _1) :
    Minus(_1, 1, _4) &
    Minus(_1, 2, _5) &
    $fib(_4, F1) & $fib(_5, F2) &
    Plus(F1, F2, _6) & Unify(_6, _2).
```

Figure 9: Intermediate compilation of the fib/2 predicate.

invoked 88, 89, 176 and 88 times on both occasions. Note also that the recursive clause of fib/2 requires nine variables whereas the recursive clause of $fib/2 uses seven variables. Furthermore, the $fib/2 clause has more scope for introducing local variables since five of its variables can be allocated to a stack. For the fib/2 clause, only three of its nine variables are local.

It is important to realise that although schedule analysis guarantees that a minimum number of suspensions are created for fib/2, it does not ensure that the suspension count is actually reduced. Schedulers often employ the heuristic that data-dependencies tend to flow left-to-right among the atoms of a clause to avoid creating unnecessary suspensions. In particular, for the fib/2 predicate, all the data-dependencies flow left-to-right, and therefore the scheduling heuristic minimises the number of suspensions. This does not deny the usefulness of schedule analysis since the use of threads enables other scheduling optimisations to be applied. For instance, threads permit the run-queue to be accessed less frequently and also enable a reduction in the number of arguments which are copied to and from a stack [5].

Nevertheless, schedule analysis cannot significantly improve performance when all threads it produces are small. This tends to occur with programs which use coroutining throughout. The application of schedule analysis to a prime sieve program, for example, only gave a marginal improvement in performance due to the instruction count for Unify/2 reducing from 275 to 247.

5 Related work

Korsloot and Tick [10] have presented some initial ideas on how to introduce sequentiality into concurrent logic programs. Like schedule analysis the aim is to recover "traditional procedural language optimisations that have previously been discarded by those implementing committed-choice languages". Korsloot and Tick derive data-dependencies by the mode algorithm described by Ueda and Morita [13] and give several examples of how the data-dependencies can be used to order the atoms of a clause. The procedure for sequentialisation is *ad hoc*, has no supporting theory, and consequently there is no guarantee that deadlock is avoided.

6 Conclusions

Schedule analysis is concerned with deducing at compile-time a partial schedule of the processes, or equivalently the body atoms of a clause, which is consistent with the behaviour of the program. It partitions the atoms of each clause into threads of totally ordered atoms which do not contradict any data-dependence of the program. Threads substitute data-flow with control-flow thereby reducing the load on a scheduler and also enabling a wealth of traditional control-flow optimisations to be applied to the program.

A framework for schedule analysis has been proposed, formulated in terms of the operational semantics for a program, which builds from the notion of a data-dependence to define a procedure for creating threads. All data-dependencies which can possibly occur between the atoms of a clause for any query, are collected together and compared, to identify those atoms of a clause which must be allocated to different threads. This gives a straightforward prescription for partitioning the atoms of a clause into threads. The threads generated by this procedure, however, in exceptional circumstances, can compromise the behaviour of the program. Thus, a safety result has been presented which states the conditions under which the work of a scheduler can be safely reduced from scheduling processes to scheduling threads. The theorem provides a way for checked the integrity of the threads so that erroneous threads can be identified and filtered out.

Deriving the data-dependencies for schedule analysis is non-trivial and will be a focus of future work. Nevertheless, a preliminary implementation suggests that schedule analysis is likely to be a useful compilation technique.

Acknowledgments

This research was supported in part by the UK Science and Engineering Research Council, Research Grant GR/H15042.

References

[1] Foster, I. & W. Winsborough (1991). "Copy Avoidance through Compile-Time Analysis and Local Reuse", *in Proceedings of the 1991 International Logic Programming Symposium.* MIT Press.

[2] Gregory, S. (1987). *Parallel Logic Programming in Parlog, The Language and its Implementation.* Addison-Wesley.

[3] Karp, R. M. (1972). *Complexity of Computer Computations*, pp. 85–103. Plenum Press.

[4] King, A. & P. Soper (1990). "Granularity Analysis of Concurrent Logic Programs", *in The Fifth International Symposium on Computer and Information Sciences*, Nevsehir, Cappadocia, Turkey.

[5] King, A. & P. Soper (1991)a. "Implementing and Optimising Threads", Technical Report CSTR 91-13, University of Southampton.

[6] King, A. & P. Soper (1991)b. "A Semantic Approach to Producer and Consumer Analysis", *in International Conference on Logic Programming Workshop on Concurrent Logic Programming*, Paris, France.

[7] King, A. & P. Soper (1992)a. "Heuristics, Thresholding and a New Technique for Controlling the Granularity of Concurrent Logic Programs", Technical Report 92-08, University of Southampton.

[8] King, A. & P. Soper (1992)b. "Ordering Optimisations for Concurrent Logic Programs", *in Proceedings of the Logical Foundations of Computer Science Symposium*, Tver, Russia. Springer-Verlag.

[9] King, A. & P. Soper (1992)c. "Schedule Analysis: a full theory, a pilot implementation and a preliminary assessment", Technical Report 92-06, University of Southampton.

[10] Korsloot, M. & E. Tick (1991). "Sequentializing Parallel Programs", *in Proceedings of the Phoenix Seminar and Workshop on Declarative Programming*, Hohritt, Sasbachwalden, Germany. Springer-Verlag.

[11] Matula, D. W., G. Marble, & J. D. Isaacson (1972). *Graph Theory and Computing*, pp. 109–122. Prentice Hall.

[12] Tiernan, J. C. (1970). "An efficient search algorithm to find the elementary circuits of a graph", *Communications of the ACM*, 13: 722–726.

[13] Ueda, K. & M. Morita (1990). "A New Implementation Technique for flat GHC", *in International Conference on Logic Programming*, pp. 3–17, Jerusalem. MIT Press.