Minimizing the overheads of dependent AND-parallelism

Peter Wang
Mission Critical Australia
novalazy@gmail.com

Zoltan Somogyi
Department of Computer Science and Software Engineering
University of Melbourne, Australia and
National ICT Australia (NICTA)
zs@unimelb.edu.au

July 7, 2011

Abstract

Parallel implementations of programming languages need to control synchronization overheads. Synchronization is essential for ensuring the correctness of parallel code, yet it adds overheads that aren't present in sequential programs. This is an important problem for parallel logic programming systems, because almost every action in such programs requires accessing variables, and the traditional approach of adding synchronization code to all such accesses is so prohibitively expensive that a parallel version of the program may run more slowly on four processors than a sequential version would run on one processor. We present a program transformation for implementing dependent AND-parallelism in logic programming languages that uses mode information to add synchronization code only to the variable accesses that actually need it.

1 Introduction

The usual motivation for parallelism is higher performance. Parallelizing imperative programs is notoriously errorprone, whereas in some declarative languages, introducing parallelism into a program can be as easy as adding a directive asking for it. Unfortunately, parallel implementations of declarative languages have a big enemy: overheads.

Parallel systems have overheads for the mechanisms used to manage the parallelism itself, as well as the overheads present in the sequential system they are based on. To get speedups over the best sequential systems, overheads must be kept to a minimum. Unfortunately, most sequential implementations of logic programming languages have quite high overheads. Most Prolog systems employ one variant or another of the Warren Abstract Machine or WAM [18], either compiling Prolog programs to WAM bytecode and then interpreting that bytecode, or by compiling the bytecode further down to C [2] or to native code [5]. Unfortunately, even with the latter approach, it is very hard to eliminate the overheads that are effectively designed into WAM instructions. For example, in the WAM, implementing `A is B + C` requires (a) following a chain of zero or more variable-to-variable references for each of `A`, `B` and `C`, (b) checking that `B` and `C` are bound to integers, (c) removing the tags from their values, (d) adding the values, (e) putting the "integer" tag on the result, (f) checking that `A` is unbound, (g) binding it to the tagged result, (h) checking to see if the binding needs to be trailed, and if yes (i) trailing the binding. This will usually require 10 to 20 instructions, when you should need only one or two ((d) and maybe (g)). For each of the nonessential steps, there are analyses that allows that step to be eliminated in a given instance, but even in the few systems that have such analyses they are nowhere near 100% effective. For example, one of the best such systems, Ciao-Prolog, averages only about half the speed of Mercury on several benchmarks [8]. The only way to *guarantee* the minimization of sequential overheads is to design a language that removes the need for them as far as possible, which can be done by moving all possible decisions to compile time. The best way to minimize overheads is to design a language that removes the need for them as far as possible, which is best done by moving all possible decisions to compile time. The only logic programming language designed with this objective is Mercury. Its implementation has consequently long been the fastest among sequential implementations of logic programming languages [14], though recently systems like YAP [4] have been catching up. This speed makes Mercury a good starting point for a parallel logic programming implementation.

Most parts of most Mercury programs are deterministic,[1] so Mercury needs AND-parallelism, not OR-parallelism, and since most goals in predicate bodies depend on the goals preceding them,[2] it needs *dependent* AND-parallelism. If you want to use this parallelism to achieve higher speeds than would be possible on one processor in *any* language (which may become the "killer application" of declarative languages when we enter the era of CPUs with 16+ cores), then you need to be very aggressive in keeping down the overheads of the synchronization required to implement the dependencies between parallel goals. Our paper presents a way to do this.

The structure of the paper is as follows. Section 2 briefly reviews the Mercury language. Sections 3 and 4 present the main contributions of the paper, our synchronization and specialization transformations respectively. Section 5 gives some performance results, while section 6 concludes with comparisons to related work.

2 Background

2.1 Mercury

The part of Mercury relevant to this paper can summarized by this abstract syntax:

pred P	: $p(x_1, \ldots, x_n) \leftarrow G$	predicates
goal G	: $x = y \mid x = f(y_1, \ldots, y_n)$	unifications
	$\mid p(x_1, \ldots, x_n) \mid x_0(x_1, \ldots, x_n)$	first and higher order calls
	$\mid$ foreign(p, *foreign code*,	
	$[x_1 : f_1, \ldots, x_n : f_n]$)	foreign code
	$\mid (G_1, \ldots, G_n) \mid (G_1 \ \& \ \ldots \ \& \ G_n)$	seq and par conjunctions
	$\mid (G_1; \ldots; G_n) \mid$ switch x $(\ldots;\ f_i : G_i; \ldots)$	disjunctions and switches
	$\mid$ (*if* G_c *then* G_t *else* G_e) $\mid$ *not* G	if-then-elses and negations
	$\mid$ *some* $[x_1, \ldots, x_n]$ G	quantifications
	$\mid$ *promise_pure* G $\mid$ *promise_semipure* G	purity assertions

[1] In the Mercury compiler, about 2% of all predicates can succeed more than once, about 22% can succeed at most once, and about 76% are guaranteed to succeed exactly once.

[2] The version of the Mercury compiler in the 11.07 release of Mercury contains 53 conjunctions with two or more expensive conjuncts. The expensive conjuncts are independent of one another in only one of these.

The atomic constructs of Mercury are unifications, (which the compiler breaks down until they have at most one function symbol each), plain first-order calls, higher-order calls, and calls to predicates defined by code in a foreign language (usually C). (To allow inlining, the representation of the last construct includes not just the name of the predicate being called but also the foreign code that is predicate's definition and the mapping from the Mercury variables that are the call's arguments to the names of the variables that stand for them in the foreign code.)

The composite constructs include sequential disjunctions, if-then-elses, negations and existential quantifications[3] These should all be self-explanatory. A parallel conjunction is a conjunction in which all the conjuncts execute in parallel. They may be independent or dependent; if dependent, the dependencies must be respected. (The mechanism for enforcing the dependencies is the subject of this paper.) A switch is a disjunction in which each disjunct unifies the same variable, whose value is known, with a different function symbol. Switches in Mercury are thus analogous to switches in languages like C. We explain purity assertions below.

Mercury has a strong mode system. The mode system classifies each argument of each predicate as either input or output; there are exceptions, but they are not relevant to this paper. If input, the caller must pass a ground term for that argument. If output, the caller must pass a distinct free variable, which the predicate will instantiate to a ground term. A predicate may have more than one mode; we call each one a *procedure*. The Mercury compiler generates different code for different procedures; in fact, different procedures are handled as separate entities by all parts of the compiler after mode checking. The compiler is responsible for reordering conjuncts in both sequential and parallel conjunctions as necessary to ensure that for each variable shared between conjuncts, the goal that generates the value of the variable (the *producer*) comes before all the goals that use this value (the *consumers*). This means that for each variable in each procedure, the compiler knows exactly where in that procedure that variable gets grounded.

The mode system both establishes and requires some invariants.

- The conjunction invariant says that on any execution path consisting of conjoined goals, each variable that is consumed by any one of the goals is produced by exactly one goal.

- The branched goal invariant says that in disjunctions, switches and if-then-elses, the goal types that contain alternative branches of execution, each branch of execution must produce the exact same set of variables that are visible from outside the branched goal, with one exception: a branch of execution that cannot succeed may produce a subset of this set.

- Negated goal invariant: A negated goal may not bind any variable that is visible to goals outside it, and the condition of an if-then-else may not bind a variable that is visible anywhere except in the then-part of that if-then-else.

Each procedure and goal has a determinism, which puts limits on the number of its possible solutions in the absence of infinite loops and exceptions. A determinism may impose a lower limit of one solution, and it may impose an upper limit of or one solution. Goals with

[3] (The abstract syntax does not include universal quantification: they are allowed at the source level, but the compiler turns $\forall X : G$ into *not* $\exists X$: *not* G.

determinism *det* succeed exactly once, *semidet* goals at most once, *multi* goals at least once, while *nondet* goals may succeed any number of times. Each procedure's mode declaration typically declares the procedure's determinism, though if this is omitted, the compiler can infer it.

Mercury divides goals into three categories:

- pure goals have no side effects and their outputs do not depend on side effects;
- semipure goals' outputs may depend on side effects, but have none themselves;
- impure goals may have side effects.

Semipure and impure predicates have to be declared as such, and calls to them must be prefaced with either `impure` or `semipure`, whichever is appropriate. The vast majority of Mercury code is pure, with impure and semipure code confined to a few places where they are used to implement pure interfaces. (For example, the implementation of the all-solutions predicates in the Mercury standard library uses impure and semipure code.) Programmers can wrap a `promise_pure` or `promise_semipure` scope around a goal to promise to the compiler (and to human readers) that the goal itself is pure or semipure respectively, even though some of the code inside the goal may be less pure.

Before we started this work, Mercury already supported independent AND-parallelism [3]. This meant that programmers who wanted to use parallelism could indicate *which* conjunctions should be executed in parallel by writing the parallel connective '&' instead of the sequential connective ',' between the conjuncts. The compiler then checked that the code satisfied the restrictions imposed on parallel conjunctions. The first restriction required all conjuncts in parallel conjunctions to be det, and the second required them to be all independent: no conjunct could produce any variable consumed by any other conjunct. The reason for the first restriction is that while there are ways to run nondeterministic conjuncts in parallel, the overheads of the algorithms required to assure correctness in such cases make it almost impossible to achieve speedups, even on several cores, over the best possible sequential version of the program at hand. We therefore keep the first restriction, even though our work obviously removes the second.

3 Synchronization for dependent AND-parallelism

Our main objective is to minimize the cost of synchronization by incurring it only when it is truly needed. In Mercury, this is relatively easy to do, due to the compiler's complete knowledge of the data flow in every predicate. (This was one of the design criteria for the language, even though the original implementation was sequential.) Given two or more conjuncts in a conjunction, the compiler knows not only which variables they have in common (i.e. which variables they *share*), but for each shared variable, it also knows which conjunct produces the variable and which conjuncts consume it.

The main parts of our implementation are the synchronization transformation (described in this section) and the specialization transformation (in section 4) . They both operate on a representation of the source code, which is based on the Mercury abstract syntax.

The synchronization transformation processes each parallel conjunction in the program independently. For each parallel conjunction, it computes the set of variables that need synchronization because they are produced by one conjunct and consumed by later conjuncts.

```
:- pred p(int, int, int).
:- mode p(in, in, out) is det.

% before transformation
p(A, B, C) :-
  (
    q(A, D),     % produces D
    r(D, E)      % produces E
  ) & (
    s(B, F),     % produces F
    t(D, F, G)   % produces G
  ),
  C = D + E + G. % produces C
```

```
% after transformation
p(A, B, C) :-
  promise_pure (
    new_future(FutureD),
    (
      q(A, D),    % produces D
      impure signal_future(FutureD, D),
      r(D, E)     % produces E
    ) & (
      s(B, F),    % produces F
      wait_future(FutureD, D'),
      t(D', F, G) % produces G
    ),
    C = D + E + G % produces C
  ).
```

Figure 1: Example of synchronizing a parallel conjunction

For each such variable, it creates a new variable that we call a *future*. It then modifies all conjuncts to use these futures to synchronize all accesses to the shared variables.

We will introduce the synchronization transformation using the example in figure 1. The body of `p` contains a parallel conjunction, whose first conjunct is the sequential conjunction `q(A, D), r(D, E)` and whose second is the sequential conjunction `s(B, F), t(D, F, G)`. The comments indicate the modes of the calls.

The basic idea of our work is that

- the producer of a shared variable will signal the availability of a value for the shared variable *immediately after* that value has been computed, and

- all consumers of a shared variable will wait for the availability of a value for the shared variable *immediately before* they use that value for the first time.

No consumer ever accesses the shared variable directly. Instead, each consumer waits on the future, and creates its own copy (`D'` in this case).

Note that once a producer generates a value for a shared variable, that binding is stable; it will never be retracted. This is because we require producers to be det, and bindings can be retracted only by backtracking. Shallow backtracking occurs when the condition of an if-then-else fails. However, the condition is not allowed to bind any variables visible from outside the if-then-else. Deep backtracking can occur only in nondet or multi code. Any such code inside the producer must either generate no outputs, or must be inside an all-solutions predicate. Again, neither of these generates any retractable bindings for variables visible from the outside.

The call to `signal_future` is deterministic but generates no outputs. The compiler can optimize away such computations if they are pure. Our transformation marks the call to `signal_future` impure to tell the rest of the compiler that it has a side-effect that must not be optimized away. Since the call to `wait_future` is the producer of the renamed version of the shared variable that is input to the consumer, it is safe from being deleted even as pure code.

Normally, any code that contains impure code is itself impure, but here impure code is only part of the implementation of the original pure goal. That is why we surround the implementation of the parallel conjunction with an assertion promising the original purity of the code; the impurity inside the promise_pure scope cannot be seen from outside the scope.

The data structures we use for this synchronization are the futures. (We got the name from MultiLisp [6]; our futures resemble theirs.) A future consists of these four fields:

- a boolean that says whether the future has been signalled yet;
- the value of the variable, if it has been signalled;
- a list of the contexts suspended waiting for this future, if it has not;
- a mutex to allow only one engine to access the future at a time.

The implementations of the three operations on futures are quite straightforward.

- `new_future` allocates a new future on the heap, and initializes its fields to the obvious values: not yet signalled, no waiting contexts.
- `wait_future` locks the mutex. If the future has been signalled, it picks up the its value and unlocks the mutex. Otherwise, the thread suspends itself on the future, unlocking the mutex. It will pick up the value when it resumes.
- `signal_future` locks the mutex, sets the value of the shared variable, records that the value is now available, makes all the threads suspended on the future runnable, and unlocks the mutex.

Note that `wait_future` acts as the producer for consumers (it *generates* the value of the variable) while `signal_future` acts as the consumer for producers (it *consumes* the value of the variable). The signal operation thus takes two inputs, the future and the value, and does not produce anything. It is also deterministic: it will succeed exactly once. The compiler can optimize away such computations if they are pure. Since that would cause the corresponding wait operations to wait forever, we mark calls to `signal_future` as impure code, which the compiler is not allowed to optimize away.

The synchronization transformation traverses the bodies of all procedures looking for parallel conjunctions. When it finds one, it invokes the algorithm in figure 2. In our algorithms all variable names start with an upper case letter, names starting with lower case letters represent the names of functions or data constructors, square parentheses represent lists as in Prolog syntax, and ++ means list concatenation. The `seq_conj` and `par_conj` data constructors create conjunctions of the named kinds out of a list of conjuncts. For example, the parallel conjunction in figure 1 would be denoted `par_conj([seq_conj([Q, R]), seq_conj([S, T])])`, where `Q`, `R`, `S` and `T` each stand for the call to the named predicate.

If none of the conjuncts consumes a variable produced by one of the other conjuncts, then the conjunction represents independent AND-parallelism, and requires no synchronization (beyond the barrier at the end, which we will ignore from now on). Otherwise, the conjunction is dependent and needs synchronization using futures. We create one future for each shared variable; `FutureMap` maps each shared variable to the variable that holds its future. The goals that create and initialize the futures (`AllocGoals`) will all run before execution enters the parallel conjunction (`ParConjuncts`) in the transformed code.

Figure 3 shows the algorithm that transforms each parallel conjunct to ensure that the conjunct waits for each shared variable it consumes before its first use, and that it signals the

```
add_sync_par_conj(Conjuncts, Purity) returns Goal:
let SharedVars be the set of variables that are
  (a) produced by one conjunct and
  (b) consumed by one or more later conjuncts
if SharedVars = ∅:
  Goal := par_conj(Conjuncts)
else:
  FutureMap := ∅
  AllocGoals := []
  for each Var in SharedVars:
    create a new var FutureVar
    FutureMap := FutureMap ∪ {Var → FutureVar}
    AllocGoals := [new_future(FutureVar) | AllocGoals]
  ParConjuncts := []
  for each Conjunct in Conjuncts:
    ParConjuncts := ParConjuncts ++
      [add_sync_par_conjunct(Conjunct, SharedVars, FutureMap, yes)]
  AllocParGoal := seq_conj(AllocGoals ++ [par_conj(ParConjuncts)])
  switch on Purity:
    case pure:     Goal := promise_pure(AllocParGoal)
    case semipure: Goal := promise_semipure(AllocParGoal)
    case impure:   Goal := AllocParGoal
```

Figure 2: Synchronizing a parallel conjunction

availability of each shared variable it produces just after it is bound. First, the algorithm finds out which shared variables it consumes and which it produces. If there are no variables in either category, then this conjunct is independent of the others and needs no synchronization. Otherwise, we loop over the consumed and produced variables, each iteration modifying the conjunct by adding the code required to synchronize accesses to that variable.

```
add_sync_par_conjunct(Goal, SharedVars, FutureMap, Rename) returns NewGoal:
ConsumedVars := SharedVars ∩ Goal's consumed vars
ProducedVars := SharedVars ∩ Goal's produced vars
if ConsumedVars and ProducedVars are both ∅:
  NewGoal := Goal
else:
  NewGoal := Goal
  for each ConsumedVar in ConsumedVars:
    NewGoal := insert_wait_into_goal(NewGoal, ConsumedVar, FutureMap)
  for each ProducedVar in ProducedVars:
    NewGoal := insert_signal_into_goal(NewGoal, ProducedVar, FutureMap)
  if Rename:
    map each var in ConsumedVars to a fresh clone var
    apply this substitution to NewGoal
```

Figure 3: Synchronizing one parallel conjunct

We could speed up the transformation algorithm by having one traversal add the synchronization code for *all* the shared variables, not just one, but we expect that most parallel conjunctions will have only a few shared variables, so the potential speedup in the compiler itself from reducing the number of traversals is limited. We also expect that most programs will contain relatively few parallel conjunctions, so forgoing this small speedup is a small price to pay for a simpler, clearer and more maintainable algorithm.

The reason why `add_sync_par_conj` tells `add_sync_par_conjunct` to rename the variables consumed by each conjunct is to ensure that no variable is produced more than once on any one execution path (the mode system's conjunction invariant). The calls to `wait_future` bind their second argument. Since each shared variable is bound once in its producing conjunct, binding it again in one or more of its consuming conjuncts would violate this invariant. (In section 4, we will see a call to `add_sync_par_conjunct` that passes "no" for Rename, but that is in circumstances in which this invariant is not in danger.) This way, each consuming conjunct gets its own copy of the shared variable, and the code *after* the parallel conjunction gets the original version from the producing conjunct.

Figures 4, 5, 6, 7 and 8 each show part of the algorithm for inserting the call to `wait_future` on `ConsumedVar` into a goal.

The `insert_wait_into_goal` function returns `NewGoal`, an updated version of `Goal` in which every occurrence of `ConsumedVar` is preceded by code that picks up the value of `ConsumedVar` by waiting on its future. It also says whether all execution paths that lead to the success of `NewGoal` have such a wait operation on them. If they all do, then the code following the original `Goal` will be able to access `ConsumedVar` without waiting, and thus does not need to be transformed. If some or all do not, then the code after the original `Goal` will need to wait for `ConsumedVar`, and thus does need to be transformed. However, we do not want both `Goal` and the code following it to wait for and thus bind `ConsumedVar`, as this would violate the conjunction invariant. That is why, if `WaitAllPaths = false`, we replace all occurrences of `ConsumedVar` in `NewGoal` with a fresh variable. This renaming leaves the meaning of `NewGoal` unchanged. Therefore when `insert_wait_into_goal` returns `WaitAllPaths = false`, `NewGoal` will not bind `ConsumedVar`; in fact, it will not refer to it at all.

If the goal given to `insert_wait_into_goal` does not consume `ConsumedVar`, then the given goal does not need modification. If it does consume `ConsumedVar`, then the kind of modification it needs depends on what kind of goal it is, which is why the main body of

```
insert_wait_into_goal(Goal, ConsumedVar, FutureMap)
  returns <NewGoal, WaitAllPaths>:
if Goal does not consume ConsumedVar:
  NewGoal := Goal
  WaitAllPaths := false
else:
  switch on the type of Goal:
    the switch cases are shown in the following figures
    each case sets NewGoal and WaitAllPaths
  if not WaitAllPaths:
    create a clone of ConsumedVar, call it CloneVar
    apply the substitution {ConsumedVar → CloneVar} to NewGoal
```

Figure 4: Inserting waits into goals, top level

```
case unify, plain call, higher order call and foreign call:
  look up ConsumedVar in FutureMap; call the result FutureVar
  WaitGoal := wait_future(FutureVar, ConsumedVar)
  NewGoal := seq_conj([WaitGoal, Goal])
  WaitAllPaths := true
case sequential conjunction:
  let Goal be seq_conj([Conjunct_1, ..., Conjunct_n])
  NewConjuncts := []
  WaitAllPaths := false
  for i = 1 to n:
    if Conjunct_i consumes ConsumedVar:
      <NewConjunct, ConjunctWaitAllPaths> :=
        insert_wait_into_goal(Conjunct_i, ConsumedVar, FutureMap)
      if ConjunctWaitAllPaths:
        NewConjuncts := NewConjuncts ++ [NewConjunct] ++
          [Conjunct_i+1, ..., Conjunct_n]
        WaitAllPaths := true
        break out of the loop
      else:
        NewConjuncts := NewConjuncts ++ [NewConjunct]
    else:
      NewConjuncts := NewConjuncts ++ [Conjunct_i]
  NewGoal := seq_conj(NewConjuncts)
```

Figure 5: Inserting waits into goals, part 1

the insert_wait_into_goal is a switch on goal type. The base case handles atomic goals (goals that do not contain other goals): unifications, plain first order calls, higher order calls, and calls to predicates defined by foreign language code. For these goals, we insert a call to wait_future before the goal to wait for and pick up the value of the consumed variable. For unifications and calls to foreign code, this is the best we can do. For plain first order calls and maybe for higher order calls, we can wait for ConsumedVar closer to the time when it is actually needed, but we will worry about that in the next section.

insert_wait_into_goal processes the conjuncts of plain sequential conjunctions left to right. When it finds a conjunct that consumes ConsumedVar, it calls itself recursively to insert the call to wait_future into that conjunct. If all successful execution paths inside this conjunct wait for ConsumedVar, then it is guaranteed to be available by the time execution gets to the following conjuncts, so we can leave them unchanged. If only some execution paths inside this conjunct wait for ConsumedVar, then NewConjunct will wait on a renamed version of ConsumedVar, so ConsumedVar itself won't be bound when execution gets to the next conjunct. If none of the conjuncts that consume ConsumedVar wait for it on all paths, then ConsumedVar itself won't be bound by the conjunction itself, which is why in that case we return WaitAllPaths = false.

Parallel conjunctions differ from sequential conjunctions in that even if one conjunct waits for ConsumedVar on all paths, the later conjuncts cannot assume that it will be available when they need it. Each conjunct that consumes ConsumedVar thus needs to wait for it independently. However, as above, having several conjuncts all ask their call to wait_future to bind the same variable (`ConsumedVar`) would violate the invariant that no variable have more than one producer in a conjunction. A consuming conjunct actually binds `ConsumedVar` if the recursive call to `insert_wait_into_goal` on it returns `ConjunctWaitAllPaths = true`. `insert_wait_into_goal` therefore renames the occurrences of `ConsumedVar` in all such conjuncts except the first.

Given a switch on `ConsumedVar`, we obviously need to wait for the value of `ConsumedVar` before the switch. For switches on other variables, we insert the wait into the switch arms that consume `ConsumedVar`. Likewise, for disjunctions, we insert the wait into the disjuncts that consume `ConsumedVar`.

With switches, execution is guaranteed to enter at most one arm. With disjunctions, execution may enter the first disjunct, and if that fails, enter the second and then later disjuncts. It is therefore possible for one disjunct to execute `wait_future` for `ConsumedVar` after a previous disjunct has already done so. Since bindings made by one disjunct disappear when execution backtracks to the next disjunct, we do need to pick up the value of `ConsumedVar` in every disjuncts that consumes it. However, if a conjunct is *guaranteed* to wait for `ConsumedVar` *before* the first place where it can fail, then later conjuncts could pick it up using a version of `wait_future` that *assumes* that the future has already been signalled, and thus does not need any synchronization.

The condition of an if-then-else may not bind any variable that is visible from anywhere except the then-part of that if-then-else. This is to prevent the variable remaining unbound if the condition fails. That is why, if `Cond` consumes `ConsumedVar`, we have to refer to a clone of it in `NewCond`. However, if `NewCond` condition waits for `ConsumedVar` on all paths, then its value of will be available in `CloneVar` at the start of the then-part, which means that the then-part will not have to wait for it again. On the other hand, if `Cond` does not consume `ConsumedVar`, then the call to `insert_wait_into_goal` and the substitution will leave it unchanged.

```
case parallel conjunction:
  let Goal be par_conj([Conjunct_1, ..., Conjunct_n])
  NewConjuncts := []
  WaitAllPaths := false
  for i = 1 to n:
    if Conjunct_i consumes ConsumedVar:
      <NewConjunct, ConjunctWaitAllPaths> :=
        insert_wait_into_goal(Conjunct_i, ConsumedVar, FutureMap)
      if ConjunctWaitAllPaths:
        if WaitAllPaths:
          create a clone of ConsumedVar, call it CloneVar
          apply the substitution {ConsumedVar → CloneVar} to NewConjunct
        else:
          WaitAllPaths := true
      NewConjuncts := NewConjuncts ++ [NewConjunct]
    else:
      NewConjuncts := NewConjuncts ++ [Conjunct_i]
  NewGoal := par_conj(NewConjuncts)
```

Figure 6: Inserting waits into goals, part 2

```
case switch:
  let Goal be switch(SwitchVar, [Case_1, ..., Case_n])
  if SwitchVar = ConsumedVar:
    look up ConsumedVar in FutureMap; call the result FutureVar
    WaitGoal := wait_future(FutureVar, ConsumedVar)
    NewGoal := seq_conj([WaitGoal, Goal])
    WaitAllPaths := true
  else:
    NewCases := []
    WaitAllPaths := true
    for i = 1 to n:
      if Case_i's goal consumes ConsumedVar:
        <NewCaseGoal, CaseWaitAllPaths> :=
          insert_wait_into_goal(Case_i's goal, ConsumedVar, FutureMap)
        NewCase := replace Case_i's goal with NewCaseGoal
        NewCases := NewCases ++ [NewCase]
        WaitAllPaths := WaitAllPaths and CaseWaitAllPaths
      else:
        NewCases := NewCases ++ [Case_i]
        WaitAllPaths := false
    NewGoal := switch(SwitchVar, NewCases)
case disjunction:
  let Goal be disj([Disjunct_1, ..., Disjunct_n])
  NewDisjuncts := []
  WaitAllPaths := true
  for i = 1 to n:
    if Disjunct_i consumes ConsumedVar:
      <NewDisjunct, DisjunctWaitAllPaths> :=
        insert_wait_into_goal(Disjunct_i, ConsumedVar, FutureMap)
      NewDisjuncts := NewDisjuncts ++ [NewDisjunct]
      WaitAllPaths := WaitAllPaths and DisjunctWaitAllPaths
    else:
      NewDisjuncts := NewDisjuncts ++ [Disjunct]
      WaitAllPaths := false
  NewGoal := disj(NewDisjuncts)
```

Figure 7: Inserting waits into goals, part 3

We handle *not G* the same way we handle *if G then false else true*. Quantifications and purity assertions require no special handling.

In some cases, the above transformation generates code that is slightly bigger than necessary. Consider an if-then-else in which the condition starts by consuming `ConsumedVar`, and the else part needs it too. Our algorithm will insert a call to `wait_future` at the start of the condition, to make the variable available in the condition (in clone form) and in the then part, and it will insert another call to `wait_future` into the else part to make the variable available there. The latter will not have to do any actual waiting, since it cannot be reached until the former has done all the waiting that may be needed. We can dispense with the second call to `wait_future`, and its cost both in code size and in execution time, if we simply put the first

```
case if-then-else:
  let Cond, Then and Else be the if-then-else's component goals
  <NewCond, CondWaitAllPaths> :=
    insert_wait_into_goal(Cond, ConsumedVar, FutureMap)
  create a clone of ConsumedVar, call it CloneVar
  apply the substitution {ConsumedVar → CloneVar} to NewCond
  if CondWaitAllPaths:
    NewThen := seq_conj([ConsumedVar := CloneVar, Then])
    ThenWaitAllPaths := true
  else:
    <NewThen, ThenWaitAllPaths> :=
      insert_wait_into_goal(Then, ConsumedVar, FutureMap)
  <NewElse, ElseWaitAllPaths> :=
    insert_wait_into_goal(Else, ConsumedVar, FutureMap)
  NewGoal := replace the relevant parts of Goal
    with NewCond, NewThen and NewElse
  WaitAllPaths := ThenWaitAllPaths and ElseWaitAllPaths
case negation:
  let NegatedGoal be the negated goal
  <NewNegatedGoal, _> :=
    insert_wait_into_goal(NegatedGoal, ConsumedVar, FutureMap)
  create a clone of ConsumedVar, call it CloneVar
  apply the substitution {ConsumedVar → CloneVar} to NewNegatedGoal
  replace the negated goal with NewNegatedGoal
  WaitAllPaths := false
case quantification and purity assertion:
  <NewSubGoal, WaitAllPaths> :=
    insert_wait_into_goal(Goal's subgoal, ConsumedVar, FutureMap)
  NewGoal := replace the subgoal in Goal with NewSubGoal
```

Figure 8: Inserting waits into goals, part 4

call *before* the if-then-else as a whole.

In general, given a branching goal (an if-then-else, switch, or disjunction), if on all possible paths of execution, the program can do only a trivial amount of work before waiting for ConsumedVar, then it is better to wait for ConsumedVar before entering the branching goal. Our implementation defines "trivial amount of work" as a sequence of zero or more unifications and calls to builtin predicates such as addition. Since calls to other predicates count as a nontrivial amount of work, if *any* execution paths make calls before needing ConsumedVar, we will wait for ConsumedVar only after the calls, even if this yields bigger code.

Inserting signals for produced variables is similar to inserting waits for consumed variables, but a bit simpler, because the mode system of Mercury enforces significantly stronger invariants for producing variables than for consuming them. For example, a negated goal can consume a variable produced outside the negation, but it cannot produce a variable consumed outside the negation. Also, it is ok for some arms of a switch, disjunction or if-then-else to consume a variable while other arms do not, but if one arm produces a variable, then *all* arms that can succeed must produce that variable. There is thus no need for SignalAllPaths.

```
insert_signal_into_goal(Goal, ProducedVar, FutureMap) returns NewGoal:
switch on the type of Goal:
  case unify, plain call, higher order call and foreign call:
    look up ProducedVar in FutureMap; call the result FutureVar
    SignalGoal := signal_future(FutureVar, ProducedVar)
    NewGoal := seq_conj([Goal, SignalGoal])
  case sequential conjunction:
    let Goal be seq_conj([Conjunct_1, ..., Conjunct_n])
    NewConjuncts := []
    for i = 1 to n:
      if Conjunct_i produces ProducedVar:
        NewConjunct :=
          insert_signal_into_goal(Conjunct_i, ProducedVar, FutureMap)
        NewConjuncts := NewConjuncts ++ [NewConjunct] ++
          [Conjunct_i+1, ..., Conjunct_n]
        break out of the loop
      else:
        NewConjuncts := NewConjuncts ++ [Conjunct]
    NewGoal := seq_conj(NewConjuncts)
  case parallel conjunction:
    let Goal be par_conj([Conjunct_1, ..., Conjunct_n])
    NewConjuncts := []
    for i = 1 to n:
      if Conjunct_i produces ProducedVar:
        NewConjunct :=
          insert_signal_into_goal(Conjunct_i, ProducedVar, FutureMap)
        NewConjuncts := NewConjuncts ++ [NewConjunct]
      else if Conjunct_i consumes ProducedVar:
        <NewConjunct, _> :=
          insert_wait_into_goal(Conjunct_i, ProducedVar, FutureMap)
        create a clone of ProducedVar, call it CloneVar
        apply the substitution {ProducedVar → CloneVar} to NewConjunct
        NewConjuncts := NewConjuncts ++ [NewConjunct]
      else:
        NewConjuncts := NewConjuncts ++ [Conjunct_i]
    NewGoal := par_conj(NewConjuncts)
```

Figure 9: Inserting signals into goals, part 1

```
case switch:
  let Goal be switch(SwitchVar, [Case_1, ..., Case_n])
  NewCases := []
  for i = 1 to n:
    NewCaseGoal :=
      insert_signal_into_goal(Case_i's goal, ProducedVar, FutureMap)
    NewCase := replace Case's goal with NewCaseGoal
    NewCases := NewCases ++ [NewCase]
  NewGoal := switch(SwitchVar, NewCases)
case disjunction:
  let Goal be disj([Disjunct_1, ..., Disjunct_n])
  NewDisjuncts := []
  for i = 1 to n:
    NewDisjunct :=
      insert_signal_into_goal(Disjunct_i, ProducedVar, FutureMap)
    NewDisjuncts := NewDisjuncts ++ [NewDisjunct]
  NewGoal := disj(NewDisjuncts)
case if-then-else:
  let Cond, Then and Else be the if-then-else's component goals
  NewThen := insert_signal_into_goal(Then, ProducedVar, FutureMap)
  NewElse := insert_signal_into_goal(Else, ProducedVar, FutureMap)
  NewGoal := replace the relevant parts of Goal
    with NewThen and NewElse
case negation:
  cannot happen:  negated goals cannot produce any nonlocal variables
case quantification and purity assertion:
  NewSubGoal :=
    insert_signal_into_goal(Goal's subgoal, ProducedVar, FutureMap)
  NewGoal := replace the subgoal in Goal with NewSubGoal
```

Figure 10: Inserting signals into goals, part 2

4 The specialization transformation

The algorithms in the previous section try to push each `wait_future` operation as late as they can and each `signal_future` operation as early as they can, in order to maximize parallelism by maximizing the time during which the producer and the consumers of the shared variable can all run in parallel. However, the amount of parallelism they can create is limited by procedure boundaries. Overcoming this limitation is the task of the specialization algorithm. The first step in this algorithm is the "making requests" step, which looks for places in the code where creating specialized versions of procedures can increase parallelism:

- We look for plain first-order calls preceded by one or more calls to `wait_future`, followed by one or more calls to `signal_future`, or both.

- For each of these `wait_future` operations, we check whether the waited-for variable is an input argument of the call and the callee does non-trivial work before it needs the variable. `PushedInputs` is the set of variables for which the answer is "yes".

- For each of these signal_future operations, we check whether the signalled variable is an output argument of the call and the callee does non-trivial work after it produces the variable. PushedOutputs is the set of variables for which the answer is "yes".
- If PushedInputs ∪ PushedOutputs is not empty, and we have access to the code of the callee, we replace the call, all the wait_futures on PushedInputs and all the signal_futures on PushedOutputs with a call to a specialized version of the called procedure, one in which all the replaced waits and signals will be done in the callee. We also request that this specialized version be created. This request identifies the variables in PushedInputs and PushedOutputs by their position in the call's argument list.

The specialized version of the callee will have each formal parameter corresponding to a variable in PushedInputs or PushedOutputs replaced by a newly-created variable that will hold its future. This changes the mode of the pushed output arguments to input, since a future is input even when it represents a value being output.

The second step executes each request by creating the specialized version of the request's callee. To generate the body of the new procedure, we invoke add_sync_par_conjunct from figure 3 with the callee's original code as Goal. As SharedVars, we pass the formal parameters corresponding to PushedInputs ∪ PushedOutputs, and the FutureMap we pass maps each variable in SharedVars to the future that replaces it in the list of formal parameters. add_sync_par_conjunct will push the synchronization code for each variable in SharedVars as deeply into the code of the callee as possible. Since the goal returned by add_sync_par_conjunct will be the body of the specialized procedure and thus will not be conjoined to anything, it is ok for this goal to bind the consumed variables, which is why we pass "no" as the value of Rename.

We invoke the specialization algorithm after the algorithms of section 3 have inserted synchronization operations into all the procedures in the module being compiled that contain dependent parallel conjunctions. We invoke the "making requests" step on all of these procedures, looking for opportunities for specialization, and modifying the code at each such opportunity as if the specialized version of the callee already existed. In doing so, we collect a queue of specialization requests, none of which have yet been acted upon. The rest of the specialization algorithm is a fixpoint operation. While there are specialization requests that have not yet been acted upon, the algorithm selects one such request, and

- executes the request, creating the requested specialized procedure, and then
- invokes the "making requests" step of the algorithm again on the newly created procedure to look for more specialization requests.

Some of these requests may have been already acted upon, some of them may already be in the queue, and some may be new. We add the new ones to the queue. The algorithm must terminate because the number of procedures and the number of possible specialization requests for each procedure are both finite. Note that each specialization always starts with the *original*, untransformed code of the callee; procedures created by the specialization algorithm itself are never used as the basis of further specializations.

The synchronization and specialization transformations push wait operations *almost* as late as they could possibly be pushed, and they push signal operations *almost* as early as they could possibly be pushed. There are two main reasons why in practice we cannot quite push those operations *exactly* as far they could be pushed in theory.

First, at the sites of higher order calls, we don't know the identity of the called procedure, so we cannot push any wait or signal operations into it. This limitation can be overcome through a program transformation called higher order specialization, which creates copies of predicates that take higher order values as input arguments, with each copy being specialized to one specific set of values of these arguments. Since this program transformation replaces higher order calls with first order calls, it allows waits and signals to be pushed further.

Second, even if we know the identity of the callee, we may not have access to its code. The Mercury compiler normally compiles each module of the program separately, which means it has access to the bodies only of the procedures defined the module being compiled. However, if intermodule optimization is enabled, it will also have access to the bodies of *some* of the procedures in the other modules, since in that case earlier invocations of the compiler on those other modules will have made available to it processed and checked versions of the bodies of procedures in those modules that compiler heuristics select as being likely to be profitably inlined and/or specialized. Another instance of this same problem that is unfortunately effectively impossible to overcome is that we cannot push wait or signal operation operations into foreign language code.

5 Performance evaluation

We have room to report on only one benchmark, but it should suffice to demonstrate the low overheads of our system. This benchmark is a raytracer written for the 2000 ICFP programming competition. Its top level loops over the rows of the picture, with each iteration computing the RGB values of the pixels in a chunk of consecutive rows. We have two versions of this loop. The independent version (i1) computes the pixels for these rows, (i2) recurses to compute the pixels for the remaining rows, and then (i3) adds both sets of pixels to the list of pixels so far. The parallelized conjunction is `(i1 & i2), i3`, and there is no data flow from i1 to i2. The dependent version (d1) computes the pixels for these rows, (d2) adds these pixels to the list of pixels so far, and then (d3) recurses to compute the pixels for the remaining rows, based on the updated list of the pixels so far. The parallelized conjunction is `(d1, d2) & d3`, and there *is* data flow from d1 to d2 and from d2 to d3. We tested both forms of the loop with varying chunk sizes, which divided the rows into 1, 8, 16 and 32 chunks. The table shows speedups compared to the best sequential version on our test machine, which was a Dell Optiplex 755 PC with a 2.4 GHz Intel Core 2 Quad Q6600 CPU. All tests were run ten times; the results are derived from the average of the middle eight.

	1i	8i	16i	32i	1d	8d	16d	32d
sequential	1.00	1.00	1.00	1.00	0.99	0.99	0.99	0.99
parallel, 1 CPU	0.91	1.08	1.08	1.10	0.90	1.02	1.03	1.04
parallel, 2 CPUs	N/A	1.46	1.73	1.74	N/A	2.05	2.14	2.13
parallel, 3 CPUs	N/A	2.16	2.19	2.21	N/A	2.36	2.73	2.72
parallel, 4 CPUs	N/A	2.41	2.49	2.55	N/A	2.83	3.28	3.32

The parallel version of the system needs to use a real machine register to point to thread-specific data, such as each engine's abstract machine registers. On x86s, this leaves only one real register for the Mercury abstract machine, which is why switching to the parallel version of the system can yield a 10% slowdown on one CPU. However, inevitable differences between program versions (such as in the placement of code and data) happen to allow similar speedups as well.

The independent versions on the left do not need any of the algorithms in this paper. The dependent versions on the right do, and thus incur extra synchronization costs. However, these costs must be very small, since they are *massively* outweighed by the better cache utilization resulting from appending each chunk's pixels to the pixels so far immediately after computing those pixels. (In all versions, the append is done in constant time using a cord, not a list, and all versions do the same appends and compute the same cord.)

Different rows take different times to render. Dividing the input into more chunks evens out such fluctuations. The data shows this to be worthwhile (up to a point) even though it also increases synchronization overhead. Overall, the speedup of 3.32 over the sequential version and $3.32/1.04 = 3.19$ over the 1-CPU parallel version shows that we have succeeded in keeping synchronization costs to a very low level.

6 Conclusion

Here we list the main differences between our work and previous systems [16].

- In Concurrent Prolog, Parlog and GHC, almost all variable accesses need protection either against premature binding of a variable (typical if the variable access is in a guard) or against some other part of the program trying to bind the variable at the same time (typical if the variable access is in a body). The survey [12] discusses some of the algorithms needed to implement this protection, all of which have considerable overhead, overhead that needs to be incurred on most accesses. By contrast, since in our system sequential execution is the rule and parallel execution is the exception, we incur synchronization overhead on accesses only to a very small proportion of variables; the fraction can be as low as tens out of billions.

- Even when synchronization is required, our system has lower overheads. Our mode system allows the producer of a variable to bind that variable without synchronization to protect it from competing producers. As for consumers, in our system, a sleeping task always waits for exactly one variable, while in some other systems, a sleeping task needs to be woken up if *any one* of several variables is bound [12]. An example is the ward primitive from [15].

- Unlike the DASWAM and similar parallel Prolog systems [13], our system does not allow the producer of a variable to be decided at runtime, so it does not have to keep track of which goal is the leftmost goal referring to a variable.

- Our system cannot deadlock: every shared variable has a producer, which (unless it throws an exception) cannot exit without binding the variable. Languages without mode systems (such as GHC) are very vulnerable to program bugs that leave a variable uninstantiated, leaving goals in guards waiting forever for them to be bound. Such bugs are notoriously hard to find [10].

- Our system does not require programmers to divide clauses into guard and body, much less into ask guard, tell guard and body.

- Unlike Janus [11] and Doc [7], hirata86a our system allows a variable to have more than one consumer.

- Unlike most other systems that allow multiple consumers, ours requires synchronization for consumers only up to the first one that waits for the variable on all paths.
- Unlike CIAO's `<&` operator [1], our futures are significantly simpler to implement, and support finer grained parallelism.
- Unlike Moded FGHC [17], our system supports separate compilation.

One important further advantage of our system is that it completely eliminates the need for a system for controlling the order of parallel code's interactions with the outside world, such as the one in [9]. This is because Mercury uses dummy variables called I/O states to represent states of the outside world, and models operations that interact with the outside world as predicates that consume and destroy the current I/O state and produce a new one. If two or more goals in a conjunction update I/O states, the algorithm in section 3 will ensure that the parallel version of the conjunction executes the same actions in the same order as the sequential version.

The system we have described is part of recent versions of Mercury, which are available for free download from the Mercury project's web page.

We would like to thank the Mercury team for their work on the Mercury implementation, and Tom Conway in particular for implementing independent AND-parallelism.

References

[1] Daniel Cabeza and Manuel V. Hermenegildo. Implementing distributed concurrent constraint execution in the CIAO system. In *Proceedings of the AGP'96 Joint Conference on Declarative Programming*, pages 67–78, 1996.

[2] Philippe Codognet and Daniel Diaz. wamcc: Compiling Prolog to C. In *Proceedings of ICLP 12*, pages 317–331, Kanagawa, Japan, June 1995.

[3] Thomas C. Conway. *Towards parallel Mercury.* PhD thesis, University of Melbourne, 2002.

[4] Anderson Faustino da Silva and Vítor Santos Costa. The design of the YAP compiler: an optimizing compiler for logic programming languages. *J. UCS*, 12(7):764–787, 2006.

[5] Daniel Diaz and Philippe Codognet. Design and implementation of the GNU Prolog system. *Journal of Functional and Logic Programming*, 2001(6), October 2001.

[6] Robert H Halstead. Implementation of Multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on List and Functional Programming*, pages 9–17, Austin, Texas, 1984.

[7] M. Hirata. Programming language Doc and its self-description, or x = x is considered harmful. In *Proceedings of the Third Conference of Japan Society of Software Science and Technology*, pages 69–72, 1986.

[8] J. Morales, M. Carro, and M. Hermenegildo. Improving the compilation of Prolog to C using types and determinism information. In *Proceedings of PADL 6*, pages 86–103, Dallas, Texas, June 2004.

[9] K. Muthukumar and M. Hermenegildo. Complete and efficient methods for supporting side effects in independent/restricted and-parallelism. In *Proceedings of ICLP 6*, pages 80–101, Lisbon, Portugal, June 1989.

[10] Ann E. Nicholson. Declarative debugging of the parallel logic programming language GHC. In *Proceedings of the 11th Australian Computer Science Conference*, pages 225–236, 1988.

[11] Vijay Saraswat. Janus: a step towards distributed constraint programming. In *Proceedings of the Second North American Conference on Logic Programming*, pages 431–446, Austin, Texas, October 1990.

[12] Ehud Shapiro. A family of concurrent logic programming languages. *Computing Surveys*, 21(3), September 1989.

[13] Kish Shen. Overview of DASWAM: exploitation of dependent AND-parallelism. *Journal of Logic Programming*, 29(1-3):245–293, 1996.

[14] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 26(1-3):17–64, October-December 1996.

[15] Hamish Taylor. A lingua franca for concurrent logic programming, 1989.

[16] Evan Tick. The deevolution of concurrent logic programming languages. *Journal of Logic Programming*, 23(2):89–123, May 1995.

[17] Kazunori Ueda and Masao Morita. Moded flat GHC and its message-oriented implementation technique. *New Generation Computing*, 13(1):3–43, 1994.

[18] David H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, California, October 1983.