Controlling Loops in Parallel Mercury Code

Paul Bone * Zoltan Somogyi

Department of Computing and Information Systems
University of Melbourne, Australia
and NICTA Victoria Laboratory, Australia
pbone@student.unimelb.edu.au and
zs@unimelb.edu.au

Peter Schachte

Department of Computing and Information Systems
University of Melbourne, Australia
schachte@unimelb.edu.au

Abstract

Recently we built a system that uses profiling data to automatically parallelize Mercury programs by finding conjunctions with expensive conjuncts that can run in parallel with minimal synchronization delays. This worked very well in many cases, but in cases of tail recursion, we got much lower speedups than we expected, due to excessive memory usage. In this paper, we present a novel program transformation that eliminates this problem, and also allows recursive calls inside parallel conjunctions to take advantage of tail recursion optimization. Our benchmark results show that our new transformation greatly increases the speedups we can get from parallel Mercury programs; in one case, it changes no speedup into almost perfect speedup on four cores.

Categories and Subject Descriptors D.3.2 [*Programming languages*]: Language classifications(Constraint and logic languages)

General Terms Languages, Performance

1. Introduction

A few years ago, the dominant form of progress in CPU design changed. Since then, clock speeds have stagnated, but the average number of cores per CPU chip has continued to climb. Even cheap PCs now sport dual-core CPUs, and high-end server CPUs are moving to eight cores and more. This trend is forecast to continue for the foreseeable future. Therefore if applications are to become faster or more powerful, it must be through taking advantage of multiple cores. Yet designing algorithms to effectively exploit multiple cores is notoriously difficult, and even among computation-intensive programs, few have been adapted to do so. If compilers were able to automatically parallelize programs, this could lead to a significant improvement in the utilization of available computing power.

One impediment to this goal is the control of the *granularity* of the computations to be carried out in parallel. Setting up and cleaning up after a parallel computation has a significant cost, so running small tasks in parallel actually slows the computation. In earlier work [2], we presented our approach for using feedback from program profiling to select large-grained parts of a computation that can be run in parallel with minimal synchronization. For some sorts of computation, this is able to automatically introduce parallelism to achieve close to optimal speedups. In this paper, we broaden the set of programs this technique works well for by controlling memory use for some patterns of parallel computations.

```
map_foldl(M, F, L, Acc0, Acc) :-
    (
        L = [],
        Acc = Acc0
    ;
        L = [H | T],
        (
            M(H, MappedH),
            F(MappedH, Acc0, Acc1)
        &
            map_foldl(M, F, T, Acc1, Acc)
        )
    ).
```

Figure 1. Parallel `map_foldl`

When a tail recursive parallel computation begins, it first creates a *spark* (a record of a task to be executed) for the recursive call, and then executes the rest of the computation directly. When a CPU is available to carry out a computation, it allocates stack space and begins to execute the recursive call in parallel, which likewise creates a spark and begins to execute the other goals in the recursive case, and so on. When each of these computations finish, however, it must wait for the result from its recursive call before it can produce its own result. Thus all these computations, each with its own stack, must be preserved until the base case is reached, whereupon these stacks can start to be reclaimed. A tail recursive sequential computation, which runs in constant stack space, has thus been transformed into a parallel computation that allocates a complete stack for each recursive call. This would quickly exhaust memory, so we have to place a limit on the number of stacks allocated. When this limit is reached, Mercury effectively reverts to sequential execution for the remainder of the tail recursive computation.

In this paper, we propose a transformation for tail recursive and nearly tail recursive parallel procedures to limit the number of stacks allocated for their execution. The transformation explicitly limits the number of stacks allocated to recursive calls to a small multiple of the number of available cores in the system. This transformation can also be asked to remove the dependency of a parallel loop iteration on the parent stack frame from which it was spawned, allowing the parent frame to be reclaimed before the completion of the recursive call. This allows parallel tail recursive computations to run in constant stack space. The transformation is applied after the automatic parallelization transformation, so it benefits both manually and automatically parallelized Mercury code.

Our benchmark results are very encouraging. Limiting the number of stacks not only permits deep tail recursions to take advantage

* Paul's work is supported by an Australian Postgraduate Award and a NICTA top-up scholarship.

DAMP'12, January 28, 2012, Philadelphia, PA, USA.

of multiple cores, but it also significantly improves performance. For most of our benchmarks, we get near-optimal speedups.

The structure of the remainder of this paper is as follows. Section 2 gives the background needed for the rest of the paper. Section 3 describes in more detail the problem that previously caused parallel Mercury programs with loops to use much more memory than one would expect. Section 4 describes the program transformation we have developed to control memory consumption by loops. Section 5 evaluates how our system works in practice on some benchmarks, and section 6 concludes with some related work.

2. Background

The abstract syntax of the part of Mercury relevant to this paper is:

pred P	: $p(x_1, \ldots, x_n) \leftarrow G$	predicates
goal G	: $x = y \mid x = f(y_1, \ldots, y_n)$	unifications
	$\mid p(x_1, \ldots, x_n)$	first order calls
	$\mid x_0(x_1, \ldots, x_n)$	higher order calls
	$\mid (G_1, \ldots, G_n)$	sequential conjunctions
	$\mid (G_1 \;\&\; \ldots \;\&\; G_n)$	parallel conjunctions
	$\mid (G_1; \ldots; G_n)$	disjunctions
	$\mid$ switch x $(\ldots;\ f_i : G_i; \ldots)$	switches
	$\mid (if\ G_c\ then\ G_t\ else\ G_e)$	if-then-elses
	$\mid not\ G$	negations
	$\mid some\ [x_1, \ldots, x_n]\ G$	quantifications

The atomic constructs of Mercury are unifications (which the compiler breaks down until they contain at most one function symbol each), plain first-order calls, and higher-order calls. The composite constructs include sequential and parallel conjunctions, disjunctions, if-then-elses, negations and existential quantifications. These should all be self-explanatory. A switch is a disjunction in which each disjunct unifies the same bound variable with a different function symbol.

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 as the argument. If output, the caller must pass a distinct free variable, which the predicate will instantiate to a ground term. It is possible for a predicate to have more than one mode; we call each mode of a predicate a *procedure*. The compiler generates separate code for each procedure of a predicate. The mode checking pass of 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 goals that use this value (the *consumers*). This means that for each variable in each procedure, the compiler knows exactly where that variable gets grounded.

Each procedure and goal has a determinism, which may put upper and lower bounds on the number of its possible solutions (in the absence of infinite loops and exceptions): *det* procedures succeed exactly once; *semidet* procedures succeed at most once; *multi* procedures succeed at least once; and *nondet* procedures may succeed any number of times.

The Mercury runtime system has a construct called a Mercury *engine* that represents a virtual CPU. Each engine is independently schedulable by the OS, usually as a POSIX thread. The number of engines that a parallel Mercury program will allocate on startup is configurable by the user, but it defaults to the actual number of CPUs. Another construct in the Mercury runtime system is a *context*, which represents a computation in progress. An engine may be idle, or it may be executing a context; a context can be running on an engine, or it may be suspended. When a context finishes execution, its storage is put back into a pool of free contexts. The bulk of this storage consists of the two stacks used by the Mercury abstract machine: the det stack and the nondet stack. Procedures that can succeed more than once store their frames on the nondet stack; all other procedures use the det stack. The only stack that is of interest in this paper is the det stack, whose behavior is very similar to the behavior of stacks in imperative languages. Following [7], we economize on memory by using *sparks* to represent goals that have been spawned off but whose execution has not yet been started.

```
map_foldl(M, F, L, Acc0, Acc) :-
    (
        L = [],
        Acc = Acc0
    ;
        L = [H | T],
        new_future(FutureAcc1),
        (
            M(H, MappedH),
            F(MappedH, Acc0, Acc1),
            signal_future(FutureAcc1, Acc1)
        &
            map_foldl_par(M, F, T, FutureAcc1, Acc)
        )
    ).

map_foldl_par(M, F, L, FutureAcc0, Acc) :-
    (
        L = [],
        wait_future(FutureAcc0, Acc0),
        Acc = Acc0
    ;
        L = [H | T],
        new_future(FutureAcc1),
        (
            M(H, MappedH),
            wait_future(FutureAcc0, Acc0),
            F(MappedH, Acc0, Acc1),
            signal_future(FutureAcc1, Acc1)
        &
            map_foldl_par(M, F, T, FutureAcc1, Acc)
        )
    ).
```

Figure 2. `map_foldl` with synchronization

The only parallel construct in Mercury is parallel conjunction, which is denoted $(G_1 \;\&\; \ldots \;\&\; G_n)$. All the conjuncts must be det, that is, they must all have exactly one solution. This restriction greatly simplifies the implementation, since it guarantees that there can never be any need to execute $(G_2 \;\&\; \ldots \;\&\; G_n)$ multiple times, just because G_1 has succeeded multiple times. (Any local backtracking inside G_1 will not be visible to the other conjuncts; bindings made by det code are never retracted.) However, this is not a significant limitation. Since the design of Mercury strongly encourages det code, in our experience, about 75 to 85% of all Mercury procedures are det, and most programs spend an even greater fraction of their time in det code. Existing algorithms for executing nondet code in parallel have large overheads, generating slowdowns by integer factors. Thus we have given priority to parallelizing det code, which we can do with *much* lower overhead.

The Mercury compiler implements $(G_1 \;\&\; G_2 \;\&\; \ldots \;\&\; G_n)$ by creating a data structure representing a barrier, and then spawning off $(G_2 \;\&\; \ldots \;\&\; G_n)$ as a spark. Since $(G_2 \;\&\; \ldots \;\&\; G_n)$ is itself a conjunction, it is handled the same way: the context executing it first spawns off $(G_3 \;\&\; \ldots \;\&\; G_n)$ as a spark that points to the barrier created earlier, and then executes G_2 itself. Eventually, the spawned-off remainder of the conjunction consists only of the final conjunct, G_n, and the context just executes it. The code of each conjunct synchronizes on the barrier once it has completed its job. When all conjuncts have done so, the original context will continue execution after the parallel conjunction.

Mercury's mode system allows a parallel conjunct to consume variables that are produced by conjuncts to its left, but not to its

right. This guarantees the absence of circular dependencies and hence the absence of deadlocks between the conjuncts, but it does allow a conjunct to depend on data that is yet to be computed by a conjunct running in parallel. We handle these dependencies through a source-to-source transformation [8]. The compiler knows which variables are produced by one parallel conjunct and consumed by another. For each of these shared variables, it creates a data structure called a *future* [5]. When the producer has finished computing the value of the variable, it puts the value in the future and signals its availability. When a consumer needs the value of the variable, it waits for this signal, and then retrieves the value from the future.

To minimize waiting, the compiler pushes signal operations as far to the left into the producer conjunct as possible, and it pushes wait operations as far to the right into each of the consumer conjuncts as possible. This means not only pushing them into the body of the predicate called by the conjunct, but also into the bodies of the predicates they call, with the intention being that each signal is put immediately after the primitive goal that produces the value of the variable, and each wait is put immediately before the leftmost primitive goal that consumes the value of the variable. Since the compiler has complete information about which goals produce and consume which variables, the only things that can stop the pushing process from placing the wait immediately before the value is to be used and the signal immediately after it is produced are higher order calls and module boundaries: the compiler cannot push a wait or signal operation into code it cannot identify or cannot access.

Given the `map_foldl` predicate in Figure 1, this synchronization transformation generates the code in Figure 2.

3. The main problem

As Mercury is a declarative programming language, Mercury programs make heavy use of recursion. Like the compilers for most declarative languages, the Mercury compiler optimizes tail recursive procedures into code that can run in constant stack space. Since this generally makes tail recursive computations more efficient than code using other forms of recursion, typical Mercury code makes heavy use of tail recursion in particular.

Unfortunately, tail recursive computations are not naturally handled well by Mercury's implementation of parallel conjunctions. Consider the `map_foldl` predicate in Figure 1. This code applies the map predicate `M` to each element of an input list, and then uses the fold predicate `F` to accumulate (in a left-to-right order) all the results produced by `M`. The best parallelization of `map_foldl` executes `M` and `F` in parallel with the recursive call. The programmer (or an automatic tool) can make this happen in the original sequential version of `map_foldl` by replacing the comma before the recursive call with the parallel conjunction operator `&`.

The problem is that the execution of a call to `map_foldl_par` has bad memory behavior. When a context begins execution of a call to `map_foldl`, it begins by creating a spark for the second conjunct (which contains the recursive call), and executes the first conjunct (which starts with the call to `M`). If another Mercury engine is available at that time, it will pick up and execute the spark for the recursive call, itself creating a spark for another recursive call and executing the *next* call to `M`. This will continue until all Mercury engines are in use and the newest spark for a recursive call must wait for an engine. When an engine completes execution of `M` and `F`, it posts the value of `Acc1` into `FutureAcc1`. Any computations waiting for `Acc1` will then be woken up; these will be the calls that wait for `Acc0` in the next iteration. In this case, the woken code will resume execution immediately before the call to `F` in the recursive invocation of `map_foldl_par`.

One might hope that after a spark for the recursive call has been created, and once `M` and `F` had completed execution and `Acc1` has been signalled, the context used to execute the first conjunct could be released. Unfortunately, it cannot because this context is the one that was running when execution entered the parallel conjunction, and therefore this is the context whose stacks contain the state of the computation outside the parallel conjunction. If we allowed this context to be reused, then all this state would be lost.

This means that until the base case of the recursion is reached, *every* recursive call must have its own complete execution context. Since each context contains two stacks, it can occupy a rather large amount of memory, so it is not practical to simultaneously preserve an execution context for each and every recursive call to a tail-recursive predicate. Originally, programs which bumped into this problem often ran themselves and the operating system out of memory rather quickly, because the default size of every det stack was several megabytes. To reduce the scope of the problem, we made stacks dynamically expandable, which allowed us to reduce their initial size, but programs with the problem can still run out of memory, it just takes more iterations to do so. Our runtime system prevents such crashes by imposing a global limit on the number of contexts that can be running or suspended at any point: if a context is needed to execute a spark and allocating the context would breach this limit, then the spark will not be executed. Eventually, the context that created the spark will execute it on its own stack, but this limits the remainder of the recursive computation to use only that context, so parallelism is curtailed at that point.

A much better solution is to swap the order of the conjuncts in the parallel conjunction so that the conjunct containing the recursive call is executed first. This means that we will spawn off the nonrecursive conjuncts, whose contexts *can* be freed when their execution is complete. However, since the Mercury mode system requires that the producer of a variable precede all its consumers, this is possible only if the conjuncts are independent. The approach we have taken in this paper is to spawn off the nonrecursive conjuncts, and continue execution of the recursive call without swapping the order of the conjuncts. We also directly limit the number of contexts that are used in a loop to a small multiple of the number of available CPUs. Finally, we can arrange for the inputs and outputs of the nonrecursive conjuncts to be stored outside the stack frame of a tail recursive procedure, which allows such procedures to run in fixed stack space even when executed in parallel. In the next section, we explain all of these improvements.

4. The loop control transformation

The main aim of loop control is to set an upper bound on the number of contexts that a loop may use, regardless of how many iterations of the loop may be executed, without limiting the amount of parallelism available. The loops we are concerned about are predicates that we call right recursive: predicates in which the recursive execution path ends in a parallel conjunction, whose last conjunct contains the recursive call.[1] Most parallel conjunctions that occur in recursive predicates occur in such predicates, because programmers have long tended to write loops in this way in order to benefit from tail recursion. Such predicates also tend to suffer the most from the problem we described in the previous section.

To guarantee the imposition of an upper bound on the number of contexts created during one of these loops, we associate with each loop a data structure that has a fixed number of slots, and require each iteration of the loop that would spawn off a goal to reserve a slot for the context of each spawned-off computation. This slot is marked as in-use until that spawned-off computation finishes, at which time it becomes available for use by another iteration.

This scheme requires us to use two separate predicates: the first sets up the data structure (which we call the *loop control* structure) and the second actually performs the loop. The rest of the program knows only about the first predicate; the second predicate is only ever called from the first predicate and from itself. Figure 3 shows

[1] A right recursive procedure may be tail recursive, or it may not be: the recursive call could be followed by other code either within the last conjunct, or after the whole parallel conjunction.

```
map_foldl_par(M, F, L, FutureAcc0, Acc) :-
    lc_create_loop_control(LC),
    map_foldl_par_lc(LC, M, F, L, FutureAcc0, Acc).

map_foldl_par_lc(LC, M, F, L, FutureAcc0, Acc) :-
    (
        L = [],
        % The base case.
        wait_future(FutureAcc0, Acc0),
        Acc = Acc0,
        lc_finish(LC)
    ;
        L = [H | T],
        new_future(FutureAcc1),
        lc_wait_free_slot(LC, LCslot),
        lc_spawn_off(LC, LCslot, (
            M(H, MappedH),
            wait_future(FutureAcc0, Acc0),
            F(MappedH, Acc0, Acc1),
            signal_future(FutureAcc1, Acc1),
            lc_join_and_terminate(LCslot, LC)
        )),
        map_foldl_par_lc(LC, M, F, T,
            FutureAcc1, Acc)
    ).
```

Figure 3. `map_foldl` after the loop control transformation

what these predicates look like. In section 4.1, we describe the loop control structure and the operations on it; in section 4.2, we give the algorithm that does the transformation; while in section 4.3, we discuss its interaction with tail recursion optimization.

4.1 Loop control structures

The loop control structure contains the following fields:

- An array of slots, each of which contains a boolean and a pointer. The boolean says whether the slot is free, and if it is not, the pointer points to the context that is currently occupying it. When the occupying context finishes, the slot is marked as free again, but the pointer remains in the slot to make it easier (and faster) for the next computation that uses that slot to find a free context to reuse.
- The number of slots in the array.
- A count of the number of slots that are currently in use.
- A boolean flag that says whether the loop has finished or not. It is initialized to false, and is set to true as the first step of the `lc_finish` operation.
- A possibly null pointer to the *master* context, the context that created this structure, and the context that will spawn of all of the iterations. This slot will point to the master context whenever it is sleeping, and will be null at all other times.
- A mutex that allows different engines to synchronize their accesses to the structure.

The finished flag is not strictly needed for the correctness of the following operations, but it does help the runtime system make better scheduling decisions. In our description of these operations, `LC` is a reference to the whole of a loop control structure, while `LCslot` is an index into the array of slots stored within `LC`.

`LC = lc_create_loop_control()` This operation creates a new loop control structure, and initializes its fields. The number of slots in the array in the structure will be a small multiple of the number of cores in the system. The multiplier is configurable by setting an environment variable when the program is run.

`LCslot = lc_wait_free_slot(LC)` This operation tests whether `LC` has any free slots. If it does not, the operation suspends until a slot becomes available. When some slots are available, either immediately or after a wait, the operation chooses one of the free slots, marks it in use, fills in its context pointer and returns its index. It can get the context to point to from the last previous user of the slot, from a global list of free contexts, (in both cases it gets contexts which have been used previously by computations that have terminated earlier), or by allocating a new context (which typically happens only soon after startup).

`lc_spawn_off(LC, LCslot, CodeLabel)` This operation sets up the context in the loop control slot, and then puts it on a queue of ready contexts, where any engine looking for work can find it.

`lc_join_and_terminate(LC, LCslot)` This operation marks the slot named by `LCslot` in `LC` as available again. It then terminates the context executing it, allowing the engine that was running it to look for other work.

`lc_finish(LC)` This operation is executed by the master context when we know that this loop will not spawn off any more work packages. It suspends its executing context until all the slots in `LC` become free. This will happen only when all the goals spawned off by the loop have terminated. This is necessary to ensure that all variables produced by the recursive call that are *not* signalled via futures have in fact had values generated for them. A variable generated by a parallel conjunct that is consumed by a later parallel conjunct will be signalled via a future, but if the variable is consumed only by code after the parallel conjunction, then it is made available by writing its value directly in its stack slot. Therefore such variables can exist only if the original predicate had code after the parallel conjunction. This barrier is the only barrier in the loop and it is executed just once; in comparison, the normal parallel conjunction execution mechanism executes one barrier in each iteration of the loop.

See Figure 3 for an example of how we use these operations. Note in particular that in this transformed version of `map_foldl`, the spawned-off computation contains the calls to `M` and `F`, with the main thread of execution making the recursive call. This is the first step in preserving tail recursion optimization.

4.2 The loop control transformation

Our algorithm for transforming procedures to use loop control is shown in Figures 4, 5 and 6.

Figure 4 shows the top level of the algorithm, which is mainly concerned with testing whether the loop control transformation is applicable to a given procedure, and creating the interface procedure if it is.

We impose conditions (1) and (2) because we need to ensure that every loop we start for `OrigProc` is finished exactly once, by the call to `lc_finish` we insert into its base cases. If `OrigProc` is mutually recursive with some other procedure, then the recursion may terminate in a base case of the other procedure, which our algorithm does not transform. And if `OrigProc` has some execution path on which it calls itself twice, then the second call may continue executing loop iterations after a base case reached through the first call has finished the loop.

We impose conditions (3) and (4) because the Mercury implementation does not support the parallel execution of code that is not deterministic. We do not want a recursive call to be called twice because some code between the entry point of `OrigProc` and the recursive call succeeded twice, and we do not want a recursive call to be backtracked into because some code between the recursive call and the exit point of `OrigProc` has failed. These conditions prevent both of those situations.

```
loop_control_transform(OrigProc) returns NewProcs:
  let OrigGoal be OrigProc's body
  let RecParConjs be the set of parallel conjunctions
    in OrigGoal that contain recursive calls
  if
    (1) OrigProc is directly but not mutually recursive
    (2) OrigGoal has at most one recursive call
      on all possible execution paths,
    (3) OrigGoal has determinism 'det',
    (4) no recursive call is within a disjunction,
      a scope that changes the determinism of a goal,
      a negation, or the condition of a if-then-else,
    (5) no member of RecParConjs is within
      another parallel conjunction,
    (6) every recursive call is inside
      the last conjunct of a member of RecParConjs,
    (7) every execution path through
      one of these last conjuncts
      makes exactly one recursive call
  then:
    let LC be a new variable
    let LCGoal be the call
      <lc_create_loop_control(LC)>
    let LoopProcName be a unique new predicate name
    let OrigArgs be OrigProc's argument list
    LoopArgs := [LC] ++ OrigArgs
    let CallLoopGoal be the call
      <LoopProcName(LoopArgs)>
    let OrigProc' be OrigProc with its body replaced
      by the conjunction <LCGoal, CallLoopGoal>

    LoopGoal := create_loop_goal(OrigGoal,
      OrigProcName, LoopProcName, RecParConjs, LC),
    let LoopProc be a new procedure
      with name LoopProcName, arguments LoopArgs
      and body LoopGoal
    NewProcs := [OrigProc', LoopProc]
  else:
    NewProcs := [OrigProc]
```

Figure 4. The top level of the transformation algorithm

We impose condition (5) because we do not want another instance of loop control, or an instance of the normal parallel conjunction execution mechanism, to interfere with this instance of loop control.

We impose condition (6) for two reasons. First, the structure of our transformation requires right recursive code: we could not terminate the loop in base case code if the call that lead to that code was followed by any part of an earlier loop iteration. Second, allowing recursion to sometimes occur outside the parallel conjunctions we are trying to optimize would unnecessarily complicate the algorithm. (We do believe that it should possible to extend our algorithm to handle recursive calls made outside of parallel conjunctions.)

We impose condition (7) to ensure that our algorithm for transforming base cases (in Figure 6) does not have to process goals that have already been processed by our algorithm for transforming recursive calls (in Figure 5).

If the transformation is applicable, we apply it. The transformed original procedure has only one purpose: to initialize the loop control structure. Once that is done, it passes a reference to that structure to `LoopProc`, the procedure that does the actual work.

The argument list of `LoopProc` is the argument list of `OrigProc` plus the `LC` variable that holds the reference to the loop control structure. The code of `LoopProc` is derived from the code of `OrigProc`. Some execution paths in this code include a recursive call; some do not. The execution paths that contain a recursive call are transformed by the algorithm in Figure 5; the execution paths that do not are transformed by the algorithm in Figure 6.

We start with the code in Figure 5. Due to condition (6), every recursive call in `OrigGoal` will be inside the last conjunct a parallel conjunction, and the main task of `create_loop_goal` is to iterate

```
create_loop_goal(OrigGoal, OrigProcName, LoopProcName,
  RecParConjs, LC) returns LoopGoal:
LoopGoal := OrigGoal
for RecParConj in RecParConjs do:
  let RecParConj be <Conjunct_1 & ... & Conjunct_n>
  for i := 1 to n-1:
    let LCSlot_i be a new variable
    let WaitGoal_i be the call
      <lc_wait_free_slot(LC, LCSlot_i)>
    let JoinGoal_i be the call
      <lc_join_and_terminate(LC, LCSlot_i)>
    let SpawnGoal_i be a goal
      that spawns off the sequential conjunction
      <Conjunct_i, JoinGoal_i> as a work package
    let Conjunct_i' be the sequential conjunction
      <WaitGoal_i, SpawnGoal_i>
  Conjunct_n' := Conjunct_n
  for each recursive call RecCall in Conjunct_n':
    RecCall has the form
      <OrigProcName(Args)>
    let RecCall' be the call
      <LoopProcName([LC] ++ Args)>
    replace RecCall with RecCall' in Conjunct_n'
  let Replacement be the flattened form
    of the sequential conjunction
    <Conjunct_1', ..., Conjunct_n'>
  replace RecParConj in LoopGoal with Replacement
LoopGoal := put_barriers_in_base_cases(LoopGoal,
  RecParConjs, LoopProcName, LC)
```

Figure 5. Algorithm for transforming the recursive cases

over and transform these parallel conjunctions. (It is possible that some parallel conjunctions do not contain recursive calls; `create_loop_goal` will leave these untouched.)

The main aim of the loop control transformation is to limit the number of work packages spawned off by the loop at any one time, in order to limit memory consumption. The goals we want to spawn off as work packages that other cores can pick up and execute are all the conjuncts before the final recursive conjunct. (Without loop control, we would spawn off all the *later* disjuncts.) The first half of the main loop in `create_loop_goal` therefore generates code that creates and makes available each work package only after it obtains a slot for it in the loop control structure, waiting for a slot to become available if necessary. We make the spawned-off computation free that slot when it finishes.

To implement the spawning off process, we extended the internal representation of Mercury goals with a new kind of scope. The only one shown in the abstract syntax in section 2 was the existential quantification scope, but the Mercury implementation had several other kinds of scopes already, though none of those are relevant for this paper. We call the new kind of scope the spawn-off scope, and we make `SpawnGoal_i` be a scope goal of this kind. When the code generator processes such scopes, it

- generates code for the goal inside the scope (which will end with a call to `lc_join_and_terminate`),
- allocates a new label,
- puts the new label in front of that code,
- puts this labelled code aside so that later it can be added to the end of the current procedure's code, and
- inserts into the instruction stream a call to `lc_spawn_off` that specifies that the spawned-off computation should start execution at the label of the set-aside code. The other arguments of `lc_spawn_off` come from the scope kind.

Since we allocate a loop slot `LCSlot` just before we spawn off this computation, waiting for a slot to become available if needed, and free the slot once this computation has finished executing, the number of computations that have been spawned-off by this loop

```
put_barriers_in_base_cases(LoopGoal,
  RecParConjs, LoopProcName, LC) returns LoopGoal':
if LoopGoal is a parallel conjunction in RecParConjs:
  # case 1
  LoopGoal' := LoopGoal
else if there no call to LoopProcName in LoopGoal:
  # case 2
  let FinishGoal be the call <lc_finish(LC)>
  let LoopGoal' be the sequential conjunction
    <LoopGoal, FinishGoal>
else:
  # case 3
  switch on LoopGoal's goal type:
    case LoopGoal is an if-then-else:
      let LoopGoal be <ite(C, T, E)>
      T' := put_barriers_in_base_cases(T,
        RecParConjs, LoopProcName, LC)
      E' := put_barriers_in_base_cases(E,
        RecParConjs, LoopProcName, LC)
      let LoopGoal' be <ite(C, T', E')>
    case LoopGoal is a switch:
      let LoopGoal be
        <switch(V, [Case_1, ..., Case_N])>
      for i := 1 to N:
        let Case_i be <case(FunctionSymbol_i, Goal_i>
        Goal_i' := put_barriers_in_base_cases(Goal_i,
          RecParConjs, LoopProcName, LC)
        let Case_i' be <case(FunctionSymbol_i, Goal_i'>
      let LoopGoal' be
        <switch(V, [Case_1', ..., Case_N'])>
    case LoopGoal is a sequential conjunction:
      let LoopGoal be <Conj_1, ... Conj_N>
      i := 1
      while Conj_i does not contain a call
        to LoopProcName:
        i := i + 1
      Conj_i' := put_barriers_in_base_cases(Conj_i,
        RecParConjs, LoopProcName, LC)
      let LoopGoal' be LoopGoal with
        Conj_i replaced with Conj_i'
    case LoopGoal is a quantification:
      let LoopGoal be <some(Vars, SubGoal)>
      SubGoal' := put_barriers_in_base_cases(SubGoal,
        RecParConjs, LoopProcName, LC)
      let LoopGoal' be <some(Vars, SubGoal')>
```

Figure 6. Algorithm for transforming the base cases

and which have not yet been terminated cannot exceed the number of slots in the loop control structure.

The second half of the main loop in `create_loop_goal` transforms the last conjunct in the parallel conjunction by locating all the recursive calls inside it and modifying them in two ways. The first change is to make the call actually call the loop procedure, not the original procedure, which after the transformation is non-recursive; the second is to make the list of actual parameters match the loop procedure's formal parameters by adding the variable referring to the loop control structure to the argument list. Due to condition (6), there can be no recursive call in `OrigGoal` that is left untransformed when the main loop of `create_loop_goal` finishes.

In some cases, the last conjunct may simply *be* a recursive call. In some other cases, the last conjunct may be a sequential conjunction consisting of some unifications and/or some non-recursive calls as well as a recursive call, with the unifications and nonrecursive calls usually constructing and computing some of the arguments of the recursive call. And in yet other cases, the last conjunct may be an if-then-else or a switch, possibly with other if-then-elses and/or switches nested inside them. In all these cases, due to condition (7), the last parallel conjunct will execute exactly one recursive call on all its possible execution paths.

The last task of `create_loop_goal` is to invoke the `put_barriiers_in_base_cases` function that is shown in Figure 6 to transform the base cases of the goal that will later become the body of `LoopProc`. This function recurses on the structure of `LoopGoal`, as updated by the main loop in Figure 5.

When `put_barriiers_in_base_cases` is called, its caller knows that `LoopGoal` may contain the already processed parallel conjunctions (those containing recursive calls), it may contain base cases, or it may contain both. The main if-then-else in `put_barriiers_in_base_cases` handles each of these situations in turn.

If `LoopGoal` is a parallel conjunction that is in `RecParConjs`, then the main loop of `create_loop_goal` has already processed it, and due to condition (7), this function does not need to touch it. Our objective in imposing condition (7) was to make this possible.

If, on the other hand, `LoopGoal` contains no call to `LoopProc`, then it did not have any recursive calls in the first place, since (due to condition (6)) they would all have been turned into calls to `LoopProc` by the main loop of `create_loop_goal`. Therefore this goal either *is* a base case of `LoopProc`, or it is part of a base case. In either case, we add a call to `lc_finish(LC)` after it. In the middle of the correctness argument below, we will discuss why this is the right thing to do.

If both those conditions fail, then `LoopGoal` definitely contains some execution paths that execute a recursive call, and may also contain some execution paths that do not. What we do in that case (case 3) depends on what kind of goal `LoopGoal` is.

If `LoopGoal` is an if-then-else, then we know from condition (4) that any recursive calls in it must be in the then part or the else part, and by definition the last part of any base case code in the if-then-else must be in one of those two places as well. We therefore recursively process both the then part and the else part. Likewise, if `LoopGoal` is a switch (a disjunction in which each disjunct unifies a variable known to be ground with a different function symbol, so we know that at most one disjunct may succeed), some arms of the switch may execute a recursive call and some may not, and we therefore recursively process all the arms. For both if-then-elses and switches, if the possible execution paths inside them do not involve conjunctions, then the recursive invocations of `put_barriiers_in_base_cases` will add a call to `lc_finish` at the end of each execution path that does not make recursive calls.

What if those execution paths do involve conjunctions? If `LoopGoal` is a conjunction, then we recursively transform the first conjunct that makes recursive calls, and leave the conjuncts both before and after it (if any) untouched. There is guaranteed to be at least one conjunct that makes a recursive call, because if there were not, the second condition would have succeeded, and we would never get to the switch on the goal type. We also know at most one conjunct makes a recursive call. If more than one did, then there would be an execution path through those conjuncts that would make more than one recursive call, condition (2) would have failed, and the loop control transformation would not be applicable.

Correctness argument. One can view the procedure body, or indeed any goal, as a set of execution paths that diverge from each other in if-then-elses and switches (on entry to the then or else parts and the switch arms respectively) and then converge again (when execution continues after the if-then-else or switch). Our algorithm inserts calls to `lc_finish` into the procedure body at all the places needed to ensure that every nonrecursive execution path executes such a call exactly once, and does so after the last goal in the nonrecursive execution path that is not shared with a recursive execution path. These places are the ends of nonrecursive then parts whose corresponding else parts are recursive, the ends of nonrecursive else parts whose corresponding then parts are recursive, and the ends of nonrecursive switch arms where at least one other switch arm is recursive. Condition (4) tests for recursive calls in the conditions of if-then-elses (which are rare in any case) specifically to make this correctness argument possible.

Note that for most kinds of goals, execution cannot reach case 3. Unifications are not parallel conjunctions and cannot contain calls, so if `LoopGoal` is a unification, we will execute case 2.

If LoopGoal is a first order call, we will also execute case 2, because due to condition (6), all recursive calls are inside parallel conjunctions; since case 1 does not recurse, we never get to those recursive calls. LoopGoal cannot be a higher order call, since if the body of OrigGoal contains a higher order call, we cannot rule out the original procedure being mutually recursive with another procedure through that call, and condition (1) would fail. If LoopGoal is a parallel conjunction, then it is either in RecParConj, in which case we execute case 1, or (due to condition (5)) it does not contain any recursive calls, in which case we execute case 2. Condition (4) also guarantees that we will execute case 2 if LoopGoal is a disjunction, negation, or a quantification that changes the determinism of a goal by cutting away (indistinguishable) solutions. The only other goal type for which execution may get to case 3 are quantification scopes that have no effect on the subgoal they wrap, whose handling is trivial.

We can view the execution of a procedure body that satisfies condition (2) and therefore has at most one recursive call on every execution path as a descent from a top level invocation from another procedure to a base case, followed by ascent back to the top. During the descent, each invocation of the procedure executes the part of a recursive execution path up to the recursive call; during the ascent, after each return we execute the part of the chosen recursive execution path after the recursive call. At the bottom, we execute exactly one of the nonrecursive execution paths.

In our case, conditions (5) and (6) guarantee that all the goals we spawn off will be spawned off during the descent phase. When we get to the bottom and commit to a nonrecursive execution path through the procedure body, we know that we will not spawn off any more goals, which is why we can invoke lc_finish at that point. We can call lc_finish at any point in LoopGoal that is after the point where we have committed to a nonrecursive execution path, and before the point where that nonrecursive execution path joins back up with some recursive execution paths.

The code at case 2 puts the call to lc_finish at the last allowed point, not the first, or a point somewhere in the middle. We chose to do this because after the code executing LoopProc has spawned off one or more goals one level above the base case, we expect that other cores will be busy executing those spawned off goals for rather longer than it takes this core to execute the base case. By making this core do as much useful work as possible before must suspend to wait for the spawned-off goals to finish, we expect to reduce the amount of work remaining to be done after the call to lc_finish by a small but possibly useful amount. lc_finish returns *after* all the spawned-off goals have finished, so any code placed after it (such as if lc_finish were placed at the first valid point) would be executed sequentially after the loop; where it would definitely add to the overall runtime. Therefore, we prefer to place lc_finish as late as possible, so that this code occurs before lc_finish and is executed in parallel with the rest of the loop, where it may have no effect on the overall runtime of the program; it will just put to good use what would otherwise be dead time.

We must of course be sure that every loop, and therefore every execution of any base case of LoopGoal, will call lc_finish exactly once: no more, no less. (It should be clear that our transformation never puts that call on an execution path that includes a recursive call.) Now any nonrecursive execution path through LoopGoal will share a (possibly empty) initial part and a (possibly empty) final part with some recursive execution paths. On any nonrecursive execution path, put_barriiers_in_base_cases will put the call lc_finish just before the first point where that path rejoins a recursive execution path. Since LoopProc is det (condition (3)), all recursive execution paths must consist entirely of det goals and the conditions of if-then-elses, and (due to condition (4)) cannot go through disjunctions. The difference between a nonrecursive execution path and the recursive path it rejoins must be either that one takes the then part of an if-then-else and the other takes the else part, or that they take different arms of a switch. Such an if-then-else or switch must be det: if it were semidet, LoopProc would be too, and if it were nondet or multi, then its extra solutions could be thrown away only by an existential quantification that quantifies away all the output variables of the goal inside it. But by condition (4), the part of the recursive execution path that distinguishes it from a nonrecursive path, the recursive call itself, cannot appear inside such scopes. This guarantees that the middle part of the nonrecursive execution path, which is not part of either a prefix or a suffix shared with some recursive paths, must also be det overall, though it may have nondeterminism inside it. Any code put after the second of these three parts of the execution path (shared prefix, middle, shared suffix), all three of which are det, is guaranteed to be executed exactly once.

4.3 Loop control and tail recursion

When a parallel conjunction spawns off a conjunct as a work package that other cores can pick up, the code that executes that conjunct has to know where it should pick up its inputs, where it should put its outputs, and where it should store its local variables. All the inputs come from the stack frame of the procedure that executes the parallel conjunction, and all the outputs go there as well, so the simplest solution, and the one used by the Mercury system, is for the spawned-off conjunct to do all its work in the exact same stack frame. Normally, Mercury code accesses stack slots via offsets from the standard stack pointer. Spawned-off code accesses stack slots using a special Mercury abstract machine register called the parent stack pointer, which the code that spawns off goals sets up to point to the stack frame of the procedure doing the spawning. That same spawning-off code sets up the normal stack pointer to point to the start of the stack in the context executing the work package, so any calls made by the spawned-off goal will allocate their stack frames in that stack, but the spawned-off conjunct will use the original frame in the stack of the parent context.

This approach works, and is simple to implement: the code generator generates code for spawned-off conjuncts normally, and then just substitutes the base pointer in all references to stack slots. However, it does have an obvious drawback: until the spawned-off computation finishes execution, it may make references to the stack frame of the parallel conjunction, whose space therefore cannot be reused until then. This means that even if a recursive call in the last conjunct of the parallel conjunction happens to be a tail call, it cannot have the usual tail call optimization applied to it.

Before this work, this did not matter, because the barrier synchronization needed at the end of the parallel conjunction, which had to be executed at every level of recursion except the base case, prevented tail recursion optimization anyway. However, the loop control transformation eliminates that barrier, replacing it with the single call to lc_finish in the base case. So now this limitation *does* matter in cases where all of the recursive calls in the last conjunct of a parallel conjunction are tail recursive.

If at least one call is not tail recursive, then it prevents the reuse of the original stack frame, so our system will still follow the scheme described above. However, if they all are, then our system can now be asked to follow a different approach. The code that spawns off a conjunct will allocate a frame at the start of the stack in the child context, and will copy the input variables of the spawned-off conjunct into it. The local variables of the spawned-off goal will also be stored in this stack frame. The question of where its output variables are stored is moot: there cannot *be* any output variables whose stack slots would need to be assigned to.

The reason this is true has to do with the way the Mercury compiler handles synchronization between parallel conjuncts. Any variable whose value is generated by one parallel conjunct and consumed by one or more other conjuncts in that conjunction will have a future created for it. The generating conjunct, once it has computed the value of the variable, will execute a signal_future on the variable's future to wake up any consumers that may be waiting for the value of this variable. Those consumers will get

the value of the original variable from the future, and will store that value in a variable that is local to each consumer. Since futures are always stored on the heap, the communication of bindings from one parallel conjunct to another does *not* go through the stack frame.

A variable whose value is generated by a parallel conjunct and is consumed by code after the parallel conjunction does need to have its value put into its stack slot, so that the code after the parallel conjunction can find it. However, if all the recursive calls in the last conjunct are in fact tail calls, then by definition there can be no code after the parallel conjunction. Since neither code later *in* the parallel conjunction, nor code *after* the parallel conjunction, requires the values of variables generated by a conjunct to be stored in the original stack frame, storing it in the spawned-off goal's child stack frame is good enough.

In our current system, the stack frame used by the spawned-off goal has exactly the same layout as its parent. This means that in general, both the parent and child stack frames will have some unused slots, slots used only in the *other* stack frame. This is trivial to implement, and we have not found the wasted space to be a problem. This may be because we have mostly been working with automatically parallelized programs, and our automatic parallelization tools put much effort into granularity control [2]: the rarer spawning-off a goal is, the less the effect of any wasted space.

5. Performance evaluation

We ran all our benchmarks on a Dell Optiplex 980 desktop PC with a 2.8 GHz Intel i7 860 CPU (four cores, each with two hyperthreads) running Linux 2.6.35 in 64-bit mode. Each test was run ten times with both Speedstep and TurboBoost disabled; we discarded the highest and lowest times, and averaged the rest.

We have benchmarked our system with four different programs:

mandelbrot generates a mandelbrot image. It renders the rows of the image in parallel using `map_foldl` from Figure 1.

raytracer is a raytracer written for the ICFP programming competition in 2000. Like mandelbrot, it renders the rows of the generated image in parallel, but it does not use `map_foldl`.

matrixmult multiplies two large matrices. It computes the rows of the result in parallel.

spectralnorm computes the eigenvalue of a large matrix using the power method. It has two parallel loops, both of which are executed multiple times. [2]

All these benchmarks have dependent AND-parallelism, but for two of the benchmarks, matrixmult and spectralnorm, we have created versions that use independent AND-parallelism as well. The difference between the dependent and independent versions is just the location of a unification that constructs a cell from the results of two parallel conjuncts: the unification is outside the parallel conjunction in the independent versions, while it is in the last parallel conjunct in the dependent versions.

Tables 1 and 2 presents our memory consumption and timing results respectively. In both tables, the columns list the benchmark programs, while the rows show the different ways the programs can be compiled and executed. Due to space limits, each table shows only a subset of the rows, those with the most interesting results. These subsets are different for the two tables.

In Table 1, each box has two numbers. The first reports the maximum number of contexts alive at the same time, while the second reports the maximum number of megabytes ever used to store the stacks of these contexts. In Table 2, each box has three numbers. The first is the execution time of that benchmark in seconds when it is compiled and executed in the manner prescribed by the row. The second and third numbers (the ones in parentheses) show respectively the speedup this time represents over the sequential version of the benchmark (the first row), and over the base parallel version (the second row). Some of the numbers are affected by rounding.

In both tables, the first row compiles the program without using any parallelism at all, asking the compiler to automatically convert all parallel conjunctions into sequential conjunctions. Obviously, the resulting program will execute on one core.

The second row compiles the program in a way that prepares it for parallel execution, but it still asks the compiler to automatically convert all parallel conjunctions into sequential conjunctions. The resulting executables will differ from the versions in the first row in two main ways. First, they will incur some overheads that the versions in the first row do not, overheads that are needed to support the possibility of parallel execution. The most important of these overheads is that potentially-parallel code needs a way to access thread-specific data, and therefore when a program is compiled for parallel execution, the Mercury compiler has to reserve one machine register to hold a pointer to this data, making that machine register unavailable to the rest of the Mercury abstract machine. Given the dearth of callee-save machine registers on the x86_64 (we are not set up to use caller-save registers), this can lead to very significant slowdowns: for our benchmarks, as much as 30%. The second difference is that, the garbage collector and the rest of the runtime system must be thread safe, and this incurs a runtime cost that leads to slowdowns in most cases, even when using a single core for user code. (The garbage collector uses one core in all of our tests). However, the mandelbrot program speeds up when thread safety is enabled; it does not do very much memory allocation and is therefore affected less by the overheads of thread safety in the garbage collector. Its slight speedup may be due to its different code and data layouts interacting with the cache system differently.

All the later rows compile the program for parallel execution, and leave the parallel conjunctions in the program intact. They execute the program on 1 to 4 cores (1c to 4c). The versions that execute on the same number of cores differ from each other mainly in how they handle loops. The rows marked nolc are the controls. They do not use the loop control mechanism described in this paper; instead, they rely on our system's overall limit on the number of contexts that may be created, as we described at the end of Section 3. The actual limit is the number of engines multiplied by a specified parameter, which we have set to 128 in c128 rows and to 512 in c512 rows. Our code checks this limit in a non-thread-safe manner, which means that in the presence of races, the limit can be exceeded by one or two contexts. Since different contexts can have different sized stacks, the limit is only an approximate control over memory consumption anyway, so this is an acceptable price to pay for reduced synchronization overhead.

The rows marked lcN do use our loop control mechanism, with the value of N indicating the value of another parameter we specify when the program is run. When the `lc_create_loop_control` instruction creates a loop control structure, it computes the number of slots to create in it, by multiplying the configured number of Mercury engines (each of which can execute on its own core) with this parameter. We have memory consumption results for N=1, 2 and 4. We have timing results for all of these too, but show only the results for N=2, since the timing results for N=1 and N=4 are almost identical to these. It seems that as long as we put a reasonably small limit on the number of stacks a loop control structure can use, speed is not much affected by the precise value of the limit.

The rows marked lcN,tr are like the corresponding lcN rows, but they also switch on tail recursion preservation in the two benchmarks (mandelbrot and raytracer) whose parallel loops are naturally tail recursive. The implementation of parallelism without loop control destroys this tail recursion, and so does loop control unless we ask it to preserve it. That means that mandelbrot and raytracer use tail recursion in all the test setups except for the parallel, non-loop control ones, and loop control ones without tail recur-

[2] Spectralnorm was donated by Chris King, see `http://adventuresinmercury.blogspot.com/search/label/parallelization`. We have modified Chris' code slightly.

	mandelbrot		mmult-depi		mmult-indep		raytracer		spectral-dep		spectral-indep	
seq	1	0.62	1	0.62	1	0.62	1	0.62	1	0.62	1	0.62
par, no &	1	0.62	1	0.62	1	0.62	1	0.62	1	0.62	1	0.62
par, &, 1c, nolc, c128	1	0.62	1	0.62	1	0.62	1	0.62	1	1.12	1	1.12
par, &, 1c, nolc, c512	1	0.62	1	0.62	1	0.62	1	0.62	1	1.12	1	1.12
par, &, 1c, lc1	2	1.25	2	1.25	n/a	n/a	2	1.25	2	1.75	n/a	n/a
par, &, 1c, lc2	3	1.88	3	1.88	n/a	n/a	3	1.88	3	2.38	n/a	n/a
par, &, 1c, lc4	5	3.12	5	3.12	n/a	n/a	5	3.12	5	3.62	n/a	n/a
par, &, 2c, nolc, c128	257	160.62	257	160.62	2	1.25	257	160.62	257	161.12	2	1.75
par, &, 2c, nolc, c512	601	375.62	1025	640.62	2	1.25	1025	640.62	1025	641.12	2	1.75
par, &, 2c, lc1	4	2.50	3	1.88	n/a	n/a	4	2.50	3	2.38	n/a	n/a
par, &, 2c, lc2	6	3.75	5	3.12	n/a	n/a	6	3.75	5	3.62	n/a	n/a
par, &, 2c, lc4	10	6.25	9	5.62	n/a	n/a	10	6.25	9	6.12	n/a	n/a
par, &, 3c, nolc, c128	385	240.62	385	240.62	3	1.88	385	240.62	385	241.12	3	2.38
par, &, 3c, nolc, c512	601	375.62	1200	750.00	3	1.88	1201	750.62	1537	961.12	3	2.38
par, &, 3c, lc1	5	3.12	4	2.50	n/a	n/a	5	3.12	4	3.00	n/a	n/a
par, &, 3c, lc2	8	5.00	7	4.38	n/a	n/a	8	5.00	7	4.88	n/a	n/a
par, &, 3c, lc4	14	8.75	13	8.12	n/a	n/a	14	8.75	13	8.62	n/a	n/a
par, &, 4c, nolc, c128	513	320.62	513	320.62	4	2.50	513	320.62	513	321.12	4	3.00
par, &, 4c, nolc, c512	601	375.62	1201	750.62	4	2.50	1201	750.62	2049	1281.12	4	3.00
par, &, 4c, lc1	6	3.75	5	3.12	n/a	n/a	6	3.75	5	3.62	n/a	n/a
par, &, 4c, lc2	10	6.25	9	5.62	n/a	n/a	10	6.25	9	6.12	n/a	n/a
par, &, 4c, lc4	18	11.25	17	10.62	n/a	n/a	18	11.25	17	11.12	n/a	n/a

Table 1. Peak number of contexts used, and peak memory usage for stacks, measured in megabytes

	mandelbrot	mmult-dep	mmult-indep	raytracer	spectral-dep	spectral-indep
seq	19.37 (1.00, 0.97)	7.69 (1.00, 1.42)	7.69 (1.00, 1.42)	19.50 (1.00, 1.21)	16.07 (1.00, 1.19)	16.06 (1.00, 1.19)
par, no &	18.75 (1.03, 1.00)	10.93 (0.70, 1.00)	10.93 (0.70, 1.00)	23.55 (0.83, 1.00)	19.07 (0.84, 1.00)	19.07 (0.84, 1.00)
par, &, 1c, nolc, c128	18.74 (1.03, 1.00)	10.94 (0.70, 1.00)	10.93 (0.70, 1.00)	23.46 (0.83, 1.00)	19.30 (0.83, 0.99)	19.12 (0.84, 1.00)
par, &, 1c, nolc, c512	18.74 (1.03, 1.00)	10.94 (0.70, 1.00)	10.93 (0.70, 1.00)	23.43 (0.83, 1.00)	19.30 (0.83, 0.99)	19.12 (0.84, 1.00)
par, &, 1c, lc2	18.74 (1.03, 1.00)	10.93 (0.70, 1.00)	n/a	23.54 (0.83, 1.00)	19.30 (0.83, 0.99)	n/a
par, &, 1c, lc2, tr	18.74 (1.03, 1.00)	n/a	n/a	23.79 (0.82, 0.99)	n/a	n/a
par, &, 2c, nolc, c128	17.82 (1.09, 1.05)	9.82 (0.78, 1.11)	5.49 (1.40, 1.99)	25.68 (0.76, 0.92)	19.25 (0.83, 0.99)	9.56 (1.68, 2.00)
par, &, 2c, nolc, c512	9.60 (2.02, 1.95)	6.63 (1.16, 1.65)	5.49 (1.40, 1.99)	20.34 (0.96, 1.16)	18.54 (0.87, 1.03)	9.56 (1.68, 2.00)
par, &, 2c, lc2	9.69 (2.00, 1.94)	5.48 (1.40, 1.99)	n/a	14.14 (1.38, 1.67)	9.96 (1.61, 1.91)	n/a
par, &, 2c, lc2, tr	9.78 (1.98, 1.92)	n/a	n/a	14.04 (1.39, 1.68)	n/a	n/a
par, &, 3c, nolc, c128	13.69 (1.42, 1.37)	8.70 (0.88, 1.26)	3.72 (2.07, 2.94)	26.58 (0.73, 0.89)	19.32 (0.83, 0.99)	6.44 (2.50, 2.96)
par, &, 3c, nolc, c512	6.39 (3.03, 2.93)	4.06 (1.89, 2.69)	3.72 (2.07, 2.94)	15.40 (1.27, 1.53)	17.57 (0.91, 1.09)	6.41 (2.50, 2.97)
par, &, 3c, lc2	6.29 (3.08, 2.98)	3.68 (2.09, 2.97)	n/a	10.72 (1.82, 2.20)	6.62 (2.43, 2.88)	n/a
par, &, 3c, lc2, tr	6.31 (3.07, 2.97)	n/a	n/a	10.80 (1.81, 2.18)	n/a	n/a
par, &, 4c, nolc, c128	8.35 (2.32, 2.25)	7.55 (1.02, 1.45)	2.82 (2.73, 3.88)	26.93 (0.72, 0.87)	18.91 (0.85, 1.01)	4.85 (3.31, 3.93)
par, &, 4c, nolc, c512	4.84 (4.01, 3.88)	3.15 (2.44, 3.48)	2.82 (2.73, 3.88)	14.12 (1.38, 1.67)	16.83 (0.95, 1.13)	4.85 (3.31, 3.93)
par, &, 4c, lc2	4.74 (4.09, 3.96)	2.79 (2.75, 3.92)	n/a	9.35 (2.09, 2.52)	4.98 (3.23, 3.83)	n/a
par, &, 4c, lc2, tr	4.76 (4.07, 3.94)	n/a	n/a	9.41 (2.07, 2.50)	n/a	n/a

Table 2. Execution times measured in seconds, and speedups

sion. Since the other benchmarks are not naturally tail recursive, they won't be tail recursive however they are compiled. There are no such rows in Table 1 since the results in each lcN,tr row would be identical to the corresponding lcN row.

There are several things to note in Table 1. The most important is that when the programs are run on more than one core, switching on loop control yields a dramatic reduction in the maximum number of contexts used at any one time, and therefore also in the maximum amount of memory used by stacks. (The total amount of memory used by these benchmarks is approximately the maximum of this number and the configured initial size of the heap.) This shows that we have achieved our main objective. Without loop control, the execution of three of our four dependent benchmarks (mandelbrot, matrixmult and raytracer) require the simultaneous existence of a context for every parallel task that the program can spawn off. For example, mandelbot generates an image with 600 rows, so the original context can never spawn off more than 600 other contexts.

On one core, the nolc versions spawn off sparks, but since there is no other engine to pick them up, the one engine eventually picks them up itself, and executes them in the original context. By contrast, the lc versions directly spawn off new contexts, not sparks. This avoids the overhead of converting a spark to a context, but we can do it only because we know we won't create too many contexts.

When executing on two or more cores, mandelbrot and raytracer use one more context that one would expect. Before the compiler applies the loop control transformation, it adds the synchronization operations needed by dependent parallel conjunctions. As shown by Figure 2, this duplicates the original procedure. Only the inner procedure is recursive, so the compiler performs the loop control transformation only on it. The extra context is the conjunct spawned off by the parallel conjunction in the outer procedure.

There are several things to note in Table 2 as well. The first is that in the absence of loop control, increasing the per-engine context limit from 128 to 512 yields significant speedups for three out of four the dependent benchmarks. Nevertheless, the versions with loop control significantly outperform the versions without, even c512, for all these benchmarks except mandelbrot. On mandelbrot, c512 already gets a near-perfect speedup, yet loop control still gets a small improvement. Thus on all our dependent benchmarks, switching on loop control yields a speedup while greatly reducing memory consumption.

Overall, the versions with loop control get excellent speedups on three of the benchmarks: speedups of 3.94, 3.92 and 3.83 on four CPUs for mandelbrot, matrixmult and spectralnorm respectively.

The one apparent exception, raytracer, is very memory-allocation-intensive, because it does lots of floating point arithmetic: the Mercury backend we use always boxes floating point numbers, so each floating point operation adds a new cell to the heap. Because of this, memory bandwidth may also be an issue for it, but its bigger problem is garbage collection; for another paper, we have measured it taking 40% of the runtime when run on four CPUs. Therefore the best speedup we can hope for is $(4 \times 0.6 + 0.4)/(0.6 + 0.4) = 2.8$, and we do come close to that.

Second, loop control is crucial for getting this kind of speedup, unless you are willing to waste lots of memory. On four cores, loop control raises the speedup compared to c128 from 2.25 to 3.96 for mandelbrot, from 1.45 to 3.92 for matrixmult, from 0.87 to 2.52 for raytracer, and from 1.01 to 3.83 for spectralnorm. Those are pretty impressive improvements.

Third, for the benchmarks that have versions using independent parallelism, the independent versions are faster than the dependent versions without loop control, while there is no significant difference between the speeds of the independent versions and the dependent loop control versions. For matrix multiplication, the loop control dependent version is faster, while for spectral-norm, the independent version is faster, but in both cases the difference is small. This shows that on these benchmarks, loop control completely avoids the problems described in Section 3.

Fourth, preserving tail recursion has a mixed effect on speed: of the six relevant cases (mandelbrot and raytracer on 2, 3 and 4 cores), one case gets a slight speedup, while the others get slight slowdowns. Due to the extra copying required, this tilt towards slowdowns is to be expected. However, the effect is very small: always within 1%, and usually in the noise. (For example, spectral-indep on three cores does everything exactly the same with c512 as with c128, so the difference between 6.44s and 6.41s is just noise.) The possibility of such slight slowdowns is an acceptable price to pay for allowing parallel code to recurse arbitrarily deeply while using constant stack space.

6. Conclusion

Ever since the first parallel implementations of declarative languages in the 1980s, researchers have known that getting more parallelism out of a program than the hardware could use can be a major problem, because the excess parallelism brings no benefits, only overhead, and these overheads could swamp the speedups the system would otherwise have gotten. Accordingly, they have devised systems to throttle parallelism, keeping it at a reasonable level.

However, most throttling mechanisms we know of have been general in nature, such as granularity control systems [6]. These have similar objectives, but use totally different methods: restricting the set of places in a program *where* they choose to exploit parallelism, not changing *how* they choose to exploit it.

We know of one system that tries to preserve tail recursion even when the tail comes from a parallel conjunction. The ACE system [4] normally generates one parcall frame for each parallel conjunction, but it will flatten two or more nested parcall frames into one if runtime determinacy tests indicate it is safe to do so. While these tests usually succeed for loops, they can also succeed for other code, and (unlike our system) the ACE compiler does not identify in advance the places where the optimization may apply. The other main difference from our system is the motivation: the main motivation of this mechanism in the ACE system is neither throttling nor the ability to handle unbounded input in constant stack space, but reducing the overheads of backtracking. This is totally irrelevant for us, since our restrictions prevent any interaction between AND-parallel code and code that can backtrack.

The only work on specially loop-oriented parallelism in logic languages that we are aware of is Reform Prolog [1]. This system was not designed for throttling either, but it is more general than ours in one sense (it can handle recursion in the middle of a clause) and less general in other senses (it cannot handle parallelism in any form other than loops, and it cannot execute one parallel loop inside another). It also has significantly higher overheads than our system: it traverses the whole spine of the data structure being iterated over (typically a list) *before* starting parallel execution; in some cases it synchronizes computations by busy waiting; and it requires variables stored on the heap to have a timestamp. To avoid even higher overheads, it imposes the same restriction we do: it parallelizes only deterministic code (though the definition of "deterministic" it uses is a bit different).

The only work on loop-oriented parallelism in functional languages we know of is Sisal [3]. It shares two of Reform Prolog's limits: no parallelism anywhere except loops, and no nesting of parallel computations inside one another. Since it was designed for number crunching on supercomputers, it had to have lower overheads than Reform Prolog, but it achieved those low overheads primarily by limiting the use of parallelism to loops whose iterations are *independent* of each other, which makes the problem much easier. Similarly, while ACE Prolog supports both AND- and OR-parallelism, the only form of AND-parallelism it supports is independent.

Our system is designed to throttle loops with dependent iterations, and it seems to be quite effective. By placing a hard bound on the number of contexts that may be needed to handle a single loop, our transformation allows parallel Mercury programs to do their work in a reasonable amount of memory, and since it does so without adding significant overhead, permits them to live up to their full potential. For one of our benchmarks, loop control makes a huge difference: on four cores, it turns a speedup of 1.13 into a speedup of 3.83. It significantly improves speedups on two other benchmarks, and it even helps the fourth and last benchmark, even though that was already close to the maximum possible speedup.

The other main advantage of our system is that it allows procedures to keep exploiting tail recursion optimization. If TRO is applicable to the sequential version of a procedure, then it will stay applicable to its parallel version. Many programs cannot handle large inputs without TRO, so they cannot be parallelized at all without this capability. The previous advantage may be specific to systems that resemble the Mercury implementation, but this should apply to the implementation of every eager declarative language.

We would like to thank Chris King for allowing us to use his spectralnorm benchmark.

References

[1] Johan Bevemyr, Thomas Lindgren, and Hkan Millroth. Reform Prolog: the language and its implementation. In *In Proc. of the 10th Int'l Conference on Logic Programming*, pages 283–298. MIT Press, 1993.

[2] Paul Bone, Zoltan Somogyi, and Peter Schachte. Estimating the overlap between dependent computations for automatic parallelization. *Theory and Practice of Logic Programming*, 11(4–5):575–591, 2011.

[3] John T. Feo, David C. Cann, and Rodney R. Oldehoeeft. A report on the Sisal language project. *Journal of Parallel and Disributed Computing*, 10:349–366, 1990.

[4] Gopal Gupta and p Enrico Pontelli. Optimization schemas for parallel implementation of non-deterministic languages and systems. *Software: Practice and Experience*, 31(12):1143–1181, 2001.

[5] 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.

[6] P. Lopez, M. Hermenegildo, and S. Debray. A methodology for granularity-based control of parallelism in logic programs. *Journal of Symbolic Computation*, 22(4):715–734, 1996.

[7] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore Haskell. *SIGPLAN Notices*, 44(9):65–78, 2009.

[8] Peter Wang and Zoltan Somogyi. Minimizing the overheads of dependent AND-parallelism. In *Proceedings of the 27th International Conference on Logic Programming*, Lexington, Kentucky, 2011.