Demand-Driven Dataflow for Concurrent Committed-Choice Code

Bart Massey and Evan Tick

CIS-TR-94-03
February 1994

Abstract

Concurrent logic languages have been traditionally executed in a "greedy" fashion, such that computations are *goal-driven*. In contrast, non-strict functional programs have been traditionally executed in a "dataflow" fashion, such that computations are *demand-driven*. The latter method can be superior when allocation of resources such as memory is critical, which is usually the case for large, complex, and/or reactive programs. Specifically, demand-driven execution results in more efficient scheduling and improved termination properties. This paper describes a novel technique for demand-driven execution of concurrent logic language programs.

This paper was submitted to *International Conference on Parallel Architectures and Compilation Techniques*, Montreal, August 1994.

Department of Computer and Information Science
University of Oregon

Contents

List of Figures

1 Introduction

The difference between goal-driven and demand-driven execution of concurrent languages is a difference in execution focus: tasks or results? Goal-driven paradigms measure performance as *tasks executed per unit time* and eagerly schedule tasks with regard for little else. Demand-driven paradigms measure performance as *answers delivered per unit time* and lazily schedule tasks only when they are necessary to transform data needed to produce a result. The main advantages of goal-driven systems are simplicity of design and abundant parallelism. The main advantage of demand-driven systems is better resource allocation (e.g., memory usage) for resource-critical programs.

This latter advantage outweighs all others for programs that simply won't run otherwise. We illustrate this with an example comparing goal-driven execution of a concurrent logic (Strand [6]) program and demand-driven execution of a non-strict functional (Lazy ML [2]) program. Concurrent logic programming languages (which we will refer to as CCLs for *committed-choice languages* or *concurrent constraint languages*) are based on the idea that implicit parallelism can be exploited via synchronization on logic variables [12].

A program to find the first five odd integers by generate-and-test, written in Lazy ML and Strand, is given in Figures 1 and 2, respectively. The first program will almost immediately report the first five odd integers (in reverse order), whereas the latter one will always loop until it runs out of memory. The reason for this is that the traditional CCL execution model is goal-driven: when a clause's head tests succeed, all goals in the clause body are immediately candidates for execution. By contrast, the traditional model for non-strict functional languages is demand-driven: a function evaluation may be delayed until that function result is actually needed by the computation. Thus, the Strand program continues to `generate` more and more integers, whereas the functional program `generates` an integer only when the `test` demands one. When five odd numbers have been found, the program will immediately terminate.

The example is given in Lazy ML and Strand only to make it concrete. In general, *all* CCL implementations of which the authors are aware are goal driven. The reason for this is largely historical: CCLs used a goal-stacking model rather than an environment-stacking model in order to obtain a high degree of parallelism from simple implementations by eliminating backtracking. Existing CCL implementations usually share the following characteristics: The compiler generates "goal stacking" code that creates goal records on the heap, instead of creating environment frames on a stack (as in sequential Prolog for instance). The goal records are managed as a pool or "ready queue" from which "worker" processes[1] can extract and add tasks. All bindings are made in a shared, global name space.[2] A procedure invocation suspends when a required binding is not supplied by the caller: the required variable is linked to the suspending goal by some internal data structure, and later the goal is resumed if the variable is bound. This implementation schema has evolved over the years and has proved to be resilient, but can be quite inefficient, as discussed in Section 2.

[1] We can safely assume that one worker is associated with each processor.

[2] Both shared-memory and distributed-memory multiprocessor implementations have been built around this model, using sufficient technology, such as import/export tables, to mimic the shared name space.

```
let rec generate n = n . generate (n + 1);

let rec test n (c . cs) l =
  if (n > 5) then l else
    (if (c % 2 = 1)
      then test (n + 1) cs (c . l)
      else test n cs l);

let five_odds = test 1 (generate 1) nil;
```

Figure 1: Generate-And-Test Program in Lazy ML

```
generate( N, L ) :- true |
  N1 is N + 1,
  L := [ N | Ls ],
  generate( N1, Ls ).

test( N, _, L ) :- N > 5 |
  L = [].
test( N, [ C | Cs ], L ) :- N =< 5, C mod 2 =:= 1 |
  N1 is N + 1,
  L := [ C | Ls ],
  test( N1, Cs, Ls ).
test( N, [C | Cs], L ) :- N =< 5, C mod 2 =\= 1 |
  test( N, Cs, L ).

five_odds( L ) :- true |
  generate( N, L1 ),
  test( 1, L1, L ).
```

Figure 2: Generate-And-Test Program in Strand

As Figure 1 illustrates, programs written in Lazy ML have nice synchronization properties. Unfortunately, the the demand-driven implementation of Lazy ML and similar lazy functional languages is completely dependent on the purely functional nature of the language, and thus cannot support logical variables. Therefore, the problem of demand-driven execution of CCL programs cannot be solved by a mere embedding of a CCL in Lazy ML — a new mechanism is needed.

Much work has been put into *mode analysis* of CCL programs. In its simplest definition, a mode of a variable occurrence in a procedure is either "in" (this procedure will not bind the variable) or "out" (external procedure will not bind the variable) [16]. Note that since variables can be bound to complex terms containing variables, in general we compute the modes of *paths* through terms to variables (at the leaves). There are several mode analyzers under development to collect this information [14, 9, 10, 13], and we consider such analysis technology "a given" for this paper. We call a CCL program *fully-moded*, if (among other restrictions not relevant to this paper) there is at most one output occurrence of a variable in a clause body. We call the family of fully-moded concurrent logic programs FM for short. In an FM program, mode information always identifies the single occurrence of any variable which produces the binding for the variable (namely the body occurrence with output mode). It is this fact which makes demand-driven execution possible, thus, we limit our attention to FM in this paper.

This paper illustrates a new technique for demand-driven execution of FM programs, applicable to languages such as Strand, FGHC [12], and Janus [11]. The significance of the work is that it is the first specification of a purely demand-driven mechanism for CCLs (see discussion of related work by Ueda and Morita in Section 4). If successful, this mechanism can lead to a quantum performance improvement and will facilitate reactive programming. The paper is organized as follows. Section 2 gives an overview of the scheme, whereas Section 3 lays out the technical details. The literature is reviewed in Section 4, and the work is summarized in Section 5.

2 Demand-Driven Evaluation

Because traditional CCL implementations do not rely on mode analysis, they must add all body calls of a clause to the ready set immediately upon execution of the clause. This leads to several sources of overhead relative to a demand-driven model:

1. Body calls may be scheduled and executed even when the bindings they produce are not needed elsewhere in the problem. While the example above illustrates an extreme case of this, it is also true that even procedures which stop after producing a finite number of bindings often produce more bindings than are actually used.

2. Because producers of bindings may run arbitrarily far ahead of their consumers, resource exhaustion, particularly memory exhaustion, may cause a program to fail even though the language semantics imply its success. Thus, programmers have to be very conscious of implementation details, and often must employ complex and specialized workarounds, such as bounded buffer techniques [6], to keep producers in check.

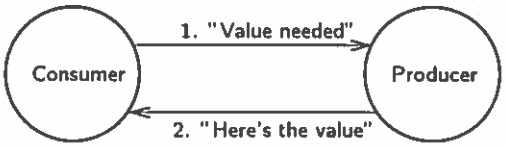

Figure 3: Transferring Control Via Continuations

3. The ready set is accessed very frequently, and may get arbitrarily large. While workarounds exist for this problem, the ready set can nonetheless become a serious performance bottleneck.

4. Because a procedure may suspend, and indeed, may suspend upon several variables, a complicated system for suspension and resumption of procedures is necessary. This mechanism typically introduces high overheads for variable binding, as well as for the suspension and resumption itself.

We propose to solve these problems by implementing a demand-driven execution scheme for FM programs. At the heart of this scheme is the use of *continuations* (e.g., [1, 4]). A continuation consists of an environment, in this case a *frame pointer*, and a program point, or *instruction pointer*. Thus, an executing worker may save its current frame pointer and instruction pointer into a continuation. Later, that continuation may be *invoked* by loading an executing worker's frame pointer and instruction pointer from the continuation's, effectively continuing execution at the point where it was left off (hence the name). Continuations have been used to great effect in programming language implementations [1], and have even been made available to the user in some programming languages [3].

A reasonable way to implement demand-driven execution of FM programs, then, would be to do something analogous to dataflow-style execution, as in Figure 3: 1) When a value is needed for execution to proceed, the consumer of that value will ask the producer for the value, by invoking a continuation in the producer. 2) The producer will supply the value, by invoking a continuation in the consumer. Indeed, this is the first principle of our design: control flow should follow dataflow.

The problem then, is how to provide the consumer with a continuation by which it may obtain a value. The key is to realize that in logic programs, the value to be obtained is inevitably *the binding of a variable.* Indeed, in traditional CCL implementations, it is exactly this fact which is used to synchronize parallel execution: an invocation will suspend when a variable the consumer wants to read has not yet been bound. Thus, it is sufficient to make the following arrangement, illustrated in Figure 4: at the time a logic variable is created, the variable's creator[3] will bind the variable to a continuation which will produce that variable's value. 1) Since the producer and consumer of a variable's binding are known to share that variable, the consumer may bind to the variable the continuation which will

[3]It is certainly possible that the creator and the producer are one and the same task.

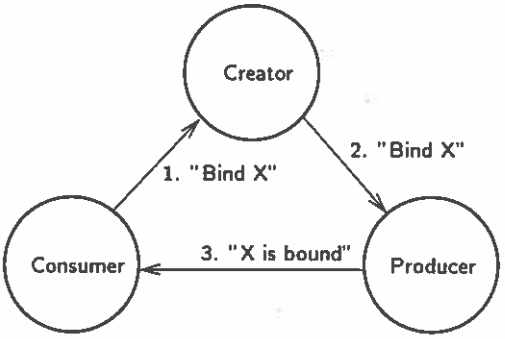

Figure 4: Continuations Bound To A Variable

consume the value produced before invoking the continuation in the creator which will eventually produce the binding. 2) The creator will arrange for a value to be bound to the variable, usually by invoking a continuation in one of its body goals. 3) The producer will bind the variable to a value before invoking the continuation which will consume the value. This is the second principle of our design: consumers and producers of a variable's binding should communicate through that variable.

The implementation should allow parallel execution, but the execution model outlined so far is sequential. To understand where the parallelism comes from, and the machinery needed to handle it, it is necessary to understand some details that have been so far avoided. First, where does the binding of a variable to a producer continuation come from? The answer is that when the variable is *created*, it is allocated in the frame of the invocation creating it, and is initialized to a continuation in this invocation. When this continuation is invoked, as noted above, there exists enough information to determine which body call will be necessary to produce a value binding for the variable.

Second, suppose that multiple consumers try to read the same variable before a binding for it is produced? This can easily happen on parallel hardware: while one worker is in the midst of producing a binding for a variable, other workers try to read the variable. The answer to the question is twofold: the variable must always be labeled with a tag indicating whether a binding is currently being produced for it, and if a binding is pending the consumer continuation must be added to a *set* of continuations bound to the variable. The producer can invoke one of the continuations directly upon finally producing a value binding, but other workers searching for work must have access to the remaining continuations, which implies a global ready set. Thus while a global ready set is apparently inevitable, a global suspend set can be avoided, by enqueuing suspended threads of execution directly on the variable causing suspension, as continuations. The third principle of our design might be stated by analogy with an epigram attributed to Einstein: avoid global objects as much as possible, but not more so.

3 Implementation Model

Having sketched the principles of operation for demand-driven execution of FM programs, we now discuss the technical details. Several preconditions must be met to make an FM program suitable for the execution model: 1) each procedure will have the head and guards of its clauses "flattened" and formed into a decision graph [8] which will select a clause body for execution; 2) each clause body will be partitioned into a "tell" part, which contains all body unifications, followed by a "call" part, containing all body calls; 3) similarly, each output argument of a clause will be categorized as either a "tell" output argument, meaning that its binding is produced locally by a tell binding, or a "call" argument, meaning that its binding is produced by a body call; 4) the binding occurrence of each variable in the clause body will be identified.

The execution model also has ramifications for FM semantics. Since execution is now demand-driven, the notion that a computation terminates when there are no ready goals no longer applies. For simplicity's sake, the computation will terminate when all output arguments of the query have been bound to ground terms.[4]

The demand-driven nature of execution allows a new guarantee about program execution: any deterministic program which *could* complete in a finite number of steps in a traditional implementation *will* complete in a finite number of steps in this implementation. Further, *any* program which will complete in a finite number of steps in this implementation will complete having created a *minimal* number of procedure invocations. These guarantees are basically a promise to the programmer that scheduling considerations are not part of programming for the demand-driven implementation.

Once this groundwork has been laid, demand-driven evaluation can be achieved by our model. The remainder of this section describes the model. The objects required by the model are described in Section 3.1. The execution algorithm of the model is outlined in Section 3.2. Finally, the detailed model and its implementation of parallelism is discussed in Section 3.3.

3.1 Objects Required by the Model

It is easiest to characterize the objects required by the model in terms of their scope. There are four kinds of state which are relevant. The *global state* is visible and accessible everywhere during execution. The *worker state* is information available only to a particular worker during its execution. *Variables* are available both to their creator and to any procedure invocation which has received them as parameters. *Frames* are available only to a particular procedure invocation. Each of these is now described in more detail.

Global State: The only global object required by the model is the ready set `ready`. This will be implemented as a tuple <`lock`, `queue`>, where `ready.lock` is a synchronizing lock, and `ready.queue` is a queue of *active messages* [5, 15]. A *message* (for short) is a tuple <`continuation`, `contents`>, where the `contents` of the message is merely

[4] If more complicated query termination conditions were required, new query syntax could be introduced to achieve them.

some value to be communicated: the `mr` register is loaded from the `contents` during message execution.

Worker State: The worker state is defined in terms of several registers, although local memory could be used if desired. These registers are:

`ip`: The "instruction pointer" points to the next instruction to be executed.

`fp`: The "frame pointer" points to the context in which to execute.

`mr`: The "message register" is used for communication across continuations.

$\texttt{ap}_0..\texttt{ap}_n$: The "argument pointers" are used to pass arguments to an invocation.

`cr`: The "call register" is used during frame creation.

Variables: A variable is a tuple <`lock`, `state`, `binding`>, where the `lock` is a standard synchronizing lock, the `state` is an element of the set {`unread`, `read`, `written`}, and the type of `binding` depends on the `state`:

- If `state` is `unread`, `binding` is a continuation.
- If `state` is `read`, `binding` is a continuation list.
- If `state` is `written`, `binding` is merely a value.

Note that the state always begins as `unread` and increases monotonically to `written`: this corresponds to the single-assignment property, and allows readers of a bound value to avoid locking the cell.

Frames: A *frame* is the environment of a particular procedure invocation, and as such, must contain all invocation-specific information. A frame is a tuple <`lock`, `commit_index`, `tobind`, `params`, `locals`, `vars`, `calls`>, as follows:

`lock`: A synchronizing lock used during frame initialization.

`commit_index`: The index of a clause which the invocation has committed to, or `nil` if the procedure has not yet committed.

`suspcount`: A count of variables which are waiting to be bound before decision graph evaluation can resume.

`tobind`: A list of *binding indices*, which will be used to restart other suspended calls to this invocation after commitment (see Section 3.3).

`params`: A save area for passed parameters of the invocation.

`vars`: A tuple of variables "created" (i.e., allocated and initialized) by the invocation but externally visible to the invocation's parent or children.

`calls`: A tuple as big as the maximum number of body calls in any clause of the procedure, whose elements are a tuple <`lock`, `callfp`>. The `calls` structure is used only after commitment: $\texttt{callfp}_i$ is either a pointer to the frame created for the i^{th} body call of the invocation, or `nil` if no frame has yet been created for this call.

`locals`: A scratch area, used for objects which will not be visible outside the invocation.

3.2 Execution Algorithm of the Model

The objects required by the execution model are best understood by considering the execution algorithm of the model. In outline, the execution mechanism is essentially that of the previous section, fleshed out below and illustrated in Figure 5. The heavy lines indicate parent-child invocation relationships, with the dashed lines indicating possible intervening invocations. The light lines indicate message passing. The entry points **read**, **resume**, **bind**, and **create** mentioned in the following description are discussed in Section 3.3.

1. An invocation, which we will call the "consumer" of a value, needs a variable to be bound in order to proceed with execution. This must be the result of the fact that the consumer needs the binding in order to commit to a clause.

2. The consumer obtains from the variable a continuation in another invocation, which we will call the "creator" of the variable being bound. This continuation will have the creator's `fp`, and the **read** entry point of the creator's procedure as the `ip`.

3. The consumer changes the `state` of the variable from `unread` to `read`, and makes the `binding` of the variable a **resume** continuation in the consumer, which will utilize the bound variable when it becomes available.

4. The consumer places a pointer to the variable in the `mr` of the worker, and invokes the **read** continuation in the creator.

5. The creator of the variable determines which body call is needed to bind the variable. It then computes a "binding index," denoting which output argument of the call is being requested, and places the binding index in `mr`.

6. If the creator has previously created the frame necessary for the body call, it loads `fp` with a pointer to this frame, and enters at the **bind** entry point of the procedure being called. Otherwise, the creator allocates a new frame, loads `fp` with a pointer to it, and enters at the **create** entry point of the procedure being called.

7. The previous step is repeated until an invocation, which we will call the "producer," actually produces a binding for the variable.

8. The producer obtains from the variable the **resume** continuation in the consumer which will consume the value.

9. The producer rebinds the variable so that its state is `written` and its `binding` is the variable's value.

10. The producer invokes the saved **resume** continuation, and the consumer uses the bound value.

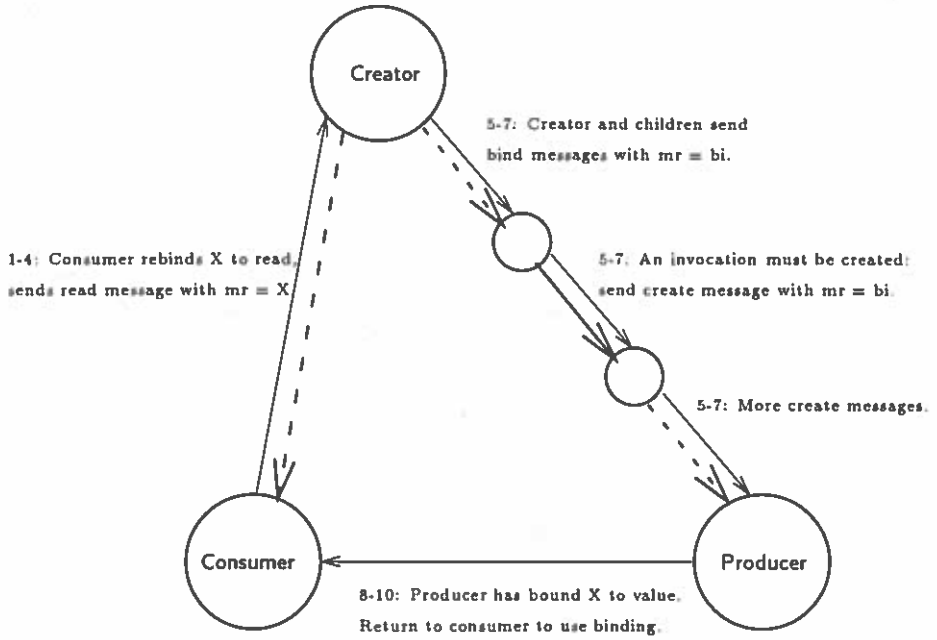

Figure 5: Control and Data Flow During Model Execution

3.3 Concurrency and Parallelism

The actual model, however, is more complicated than this! This complication is a necessary consequence of the concurrent semantics of the language. The implementation is motivated by the desire to exploit parallelism in the implementation. There are several places during execution where more than one thread of execution may need to suspend until a condition is satisfied, and all of these threads may be resumed in parallel. In addition, there is one place in which new threads of execution need to be created, and may be started in parallel.

The first place in which threads may need to suspend is the consequence of the fact that, between the time that a variables state goes to `read` and the time it goes to `written`, other procedures may also need to read the variable's value. The protocol described above assures that only one thread will try to produce the value, but it is also necessary to assure that, once the variable's value is `written`, all threads which were waiting for the value will resume. To this end, the variable's binding when in the `read` state is a *list* of continuations which should be placed on the ready queue when the variable's value becomes available. (Note that the ready queue is a message queue, not a continuation queue: According to the protocol described above, the message register `mr` is not used during resumption of suspended readers, so any value may be placed in the `mr` portion of the message.)

The second place where threads may need to suspend is a consequence of the fact that, before a procedure invocation has committed to a particular clause body, several outputs of that procedure may be requested. Thus, these threads must be suspended until commitment

is complete, then resumed. This is the purpose of the `tobind` element of the frame: it is used to hold a list of binding indices which have been requested by callers. When commitment is complete, each binding index is packaged up as the `value` portion of a message whose `ip` points to code in the procedure which will bind the variable, and whose `fp` points to the invocation's frame. These messages are then placed on the ready queue, so that all of the bindings may proceed in parallel.

New threads are created as the result of *strict* operations during commitment, i.e., operations which must have all argument values in order to proceed. An example of a *strict* operation is the arithmetic operation +/2. This operation must have *both* argument values before computing the result. Since all arguments are required, they should be produced in parallel. Thus, messages which will produce all bindings are placed on the ready queue. The frame's `suspcount` is used to keep track of when all required bindings have been obtained, at which point execution may resume.

The execution model is now complete. Due to space limitations, we omit some of the details, especially those involving locking and variable update. First, Figure 6 shows how a thread switch is performed. Each procedure has three entry points: **read**, **create**, and **bind**.

- The **read** entry point will always be reached through a continuation, when a consumer attempts to bind an `unread` variable. The message register `mr` will contain a pointer to the variable to be bound. The **read** procedure is detailed in Figure 7. **read** insures that a variable's binding will be produced, and arranges for execution to resume in the consumer thereafter.

- The **bind** entry point will be called in order to bind an output parameter of the procedure. The message register `mr` will contain a binding index indicating the parameter to be bound. The **bind** procedure is detailed in Figure 8. **read** performs body calls necessary to obtain bindings for requested output arguments.

- The **create** entry point will be called in order to create a procedure invocation (typically by the creator of a variable). The frame pointer `fp` will point to a frame which is uninitialized, except for the `lock` field, which will be locked. The message register `mr` will contain a binding index. The call register `cr` will contain a pointer to the caller's $\texttt{call}_i$ tuple. $\texttt{call}_i\texttt{.lock}$ will be locked. The **create** procedure is detailed in Figure 9. **create** handles invocation initialization, and then starts the commitment process, arranging to **bind** requested output arguments as soon as commitment completes.

To begin execution, the **create** entry point of the query is invoked, and each output argument of the query becomes the target of a **bind**. As each argument is further instantiated, **read** calls are used, until all output arguments of the query are fully ground. Then, execution simply terminates.

4 Discussion

There are a number of optimizations possible in the basic execution model. First, there are several places during execution in which messages are placed on the ready queue, and

To switch threads:

- Dequeue a message `m` from the `ready queue`.
- Load `m.mr` and `m.fp`.
- Jump to `m.ip`.

Figure 6: Thread Switch

To **read** a variable `V` passed in `mr` in a procedure `p`:

- If `mr.state` is `written`:
  - Switch threads.
- If `mr.state` is `read`:
  - Find the binding index `bi` denoted by `mr`.
  - Place `bi` in `mr`.
  - Invoke `p.bind`.

Figure 7: Procedure **read:** Request a Variable's Binding

To **bind** an output argument of a procedure `p`:

- If `mr` refers to a tell argument:
  - Switch threads.
- If `mr` refers to a call argument:
  - Find the procedure `q` and binding index `bi` denoted by `mr`.
  - Find the call index `ci` of `q`.
  - Place `bi` in `mr`.
  - If $\texttt{fp.call}_{ci}\texttt{.callfp}$ is non-nil:
    - Invoke `q.bind`.
  - If $\texttt{fp.call}_{ci}\texttt{.callfp}$ is nil:
    - Set up the arguments of a call of `q`.
    - Allocate and set up a new frame in `fp`.
    - Invoke `q.create`.

Figure 8: Procedure **bind:** Bind an Output Argument

To **create** a new frame and bindings in a procedure `p`:

- Initialize the frame's `commit_index`, `tobind`, `var`, and `call`.
- Place `fp` in `cr.fp`, marking the frame as created.
- Jump to the decision graph for `p`.

To **suspend** decision graph evaluation:

- Set `fp.suspcount` to the number of variables being suspended on.
- For each variable `V` whose binding is needed:
  - If `V.state` is `unread`:
    - Save the binding of `V` and a pointer to `V` as a message `m`.
    - Mark `V` `read`, and add a **resume** continuation to `V`'s binding.
    - Add `m` to the ready queue.
  - If `V.state` is `read`:
    - Add a resumption continuation to `V`'s binding.
- Switch threads.

To **resume** a suspended decision graph:

- Decrement `fp.suspcount`.
- If `fp.suspcount` is non-zero, switch threads.
- If `fp.suspcount` is zero, resume decision graph execution.

To **commit** to a particular clause:

- Set `fp.commit_index` to the desired clause.
- For each tell output argument `V`:
  - If `V.state` is `read`:
    - Save the binding of `V` in some temporary register `r`.
    - Bind `V` to its value.
    - For each continuation `c` in `r`:
      - Add a message containing `c` to the ready queue.
  - If `V.state` is `unread`:
    - Bind `V` to its value.
- ◇ For each call output argument `V`:
  - ◇ If `V.state` is `unread`:
    - ◇ Set `V.binding` to `p.read`.
- For each non-tell binding index `bi` in `fp.tobind`:
  - Add a message <`fp`, `p.bind`, `bi`> to the ready queue.
- Switch threads.

Figure 9: Procedure **create**: Create a Procedure Invocation

then a thread switch is done. Instead, some ready queue traffic can be avoided if, in the spirit of the original idea, one of the messages is selected for direct invocation rather than enqueuing and then immediately dequeuing it. Second, rather than adding messages to the ready queue individually, some efficiency can be gained by adding the entire batch at once. Indeed, if appropriate data structures are used, this may be almost as cheap as enqueuing an individual message. Third, although locking is not specified at this level of detail, it is apparent that much locking can be avoided due to the monotonic progression of such operations as variable binding and commitment — a thread can check to see whether a lock is necessary and avoid the lock if not. Fourth, since the **mr** is unused during resumption, it may be used by the producer to return the requested variable's binding to the consumer, avoiding an unnecessary variable reference.

The portion of the **commit** description labeled "◇" in Figure 9 is optional: it is believed that it constitutes an optimization, but empirical data must be gathered to support that claim. Without this option, attempts to bind a variable will always start at that variable's creator, and must proceed downward through a series of **bind** continuations. With the option, attempts to bind a variable will start instead at the first frame which would have to call **create** instead of **bind**, avoiding the overhead of the chain of **bind** invocations. On the other hand, with this option, every **create** procedure must perform a locked access of the variable denoted by each of its unbound call output arguments. This creates a lot of memory and lock traffic on unbound variables, which can be expensive, and may be unnecessary, inasmuch as the bindings of these variables may be produced before they are ever requested by a consumer.

There are several possible criticisms of the execution model. The most telling are:

- Nondeterministic programs may invoke procedures which do no useful work. But *any* implementation must schedule these procedure calls: only an oracle could tell which invocation will produce the bindings necessary for nondeterministic execution to proceed.

- Because of the change in termination conditions to achieve demand-driven execution, a few existing existing CCL programs might not run correctly under this model. In particular, some programs may deadlock or livelock because they expect to produce non-ground query outputs. However, the changes needed to make existing programs operable should be straightforward. Any slight incompatibility is outweighed by the fact that scheduling and throttling of producers is no longer a concern, which should make it considerably easier to write new code for this implementation.

- Compilation details such as the storage of temporaries in registers are outside the scope of this paper. However, because of the frequent switching of environments during execution, it may be difficult to fully utilize the large register set of modern CPUs for efficient temporary storage. The efficiency increases due to demand-driven execution should outweigh this loss. It is difficult to estimate the severity of this problem without empirical evidence. Thus, empirical studies should be conducted before considering solutions to the problem.

As noted in the Introduction, CCLs sacrificed backtracking in order to achieve efficiency. A consequence was the elimination of all speculative *or-parallelism*: a worker will not attempt to execute a given clause unless that clause will contribute bindings needed to answer the query. Analogously, our demand-driven implementation eliminates all speculative *and-parallelism*: a worker will not attempt to execute a given body call unless that call will contribute bindings needed to answer the query. This throttling of all speculative parallelism can lead to great efficiency in problem solution, but it may not lead to the fastest solution, as some speculative work may prove useful. It may be possible that this execution model could be extended to handle speculative work when processors become idle, e.g., as in Gregory [7], although this is a future research topic.

The only work related to our proposed scheme of which we are aware is that of Ueda and Morita [15], who describe a model which uses active messages to improve the performance of producer-consumer stream parallelism. Our use of active messages, as well as some of our fundamental philosophy, is essentially the same as theirs. However, their method has important differences from ours. First and foremost, Ueda and Morita's technique still is producer-driven rather than consumer-driven: it only attempts to optimize the overhead of producer-to-consumer communication. Thus, none of the benefits of automatic demand-driven execution accrue (although some methods of using programmer annotations to inhibit producers outrunning consumers are discussed). Second, their technique requires a sophisticated type analysis in addition to mode analysis. Finally, it is an optimization for certain limited situations only, in otherwise conventional execution. Conery [4], describes the use of continuations and messages in the implementation of an environment-stacking parallel logic programming language.

5 Conclusions

We have presented a novel execution model for flat concurrent logic programming languages (CCLs), based on mode analysis and on continuation passing utilizing shared variables. This model achieves demand-driven execution of CCLs, which at once achieves greater execution efficiency and simplifies programming. This paper serves as a schematic for those wishing to build high-performance demand-driven CCL systems. We hope to begin work soon on an exploratory implementation of the technique, in order to gain insights into its benefits and drawbacks.

References

[1] A. W. Appel, editor. *Compiling With Continuations.* Cambridge University Press, 1992.

[2] L. Augustsson and T. Johnsson. *Lazy ML User's Manual.* Available via FTP from `animal.cs.chalmers.se`, June 1993.

[3] W. Clinger and J. Rees (ed.). Revised^4 Report on the Algorithmic Language Scheme. Technical Report CIS-TR-91-25, University of Oregon, Department of Computer Science, February 1992.

[4] J. S. Conery. Continuation-Based Control in the Implementation of Parallel Logic Programs. *LISP And Symbolic Computation: An International Journal*, 7:111–134, 1994.

[5] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *International Symposium on Computer Architecture*, pages 256–266. Gold Coast, IEEE Computer Society Press, May 1992.

[6] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[7] S. Gregory. Experiments with Speculative Parallelism in Parlog. In *International Logic Programming Symposium*, pages 370–387. Vancouver B.C., MIT Press, October 1993.

[8] S. Kliger. *Compiling Concurrent Logic Programming Languages.* PhD thesis, The Weizmann Institute of Science, Rehovot, October 1992.

[9] M. Koshimura and R. Hasegawa. A Mode Analyzer for FGHC Programs in a Model Generation Theorem Prover. In *Proceedings of the* 47^{th} *Annual Convention IPS Japan*, 1993. In Japanese.

[10] B. C. Massey and E. Tick. Sequentialization of Parallel Logic Programs with Mode Analysis. In 4^{th} *International Conference on Logic Programming and Automated Reasoning*, number 698 in Lecture Notes in Artificial Intelligence, pages 205–216, St. Petersburg, July 1993. Springer-Verlag. In Press.

[11] V. A. Saraswat, K. Kahn, and J. Levy. Janus: A Step Towards Distributed Constraint Programming. In *North American Conference on Logic Programming*, pages 431–446. Austin, MIT Press, October 1990.

[12] E. Y. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[13] E. Tick. Practical Static Mode Analyses of Concurrent Logic Languages. In *International Conference on Parallel Architectures and Compilation Techniques*, Montreal, August 1994. North-Holland. Submitted.

[14] E. Tick, B. C. Massey, F. Rakoczi, and P. Tulayathun. Concurrent Logic Programs *a la Mode*. In E. Tick and G. Succi, editors, *Implementations of Logic Programming Systems*. Kluwer Academic Publishers, 1994.

[15] K. Ueda and M. Morita. Message-Oriented Parallel Implementation of Moded Flat GHC. In *International Conference on Fifth Generation Computer Systems*, pages 799–808, Tokyo, June 1992. ICOT.

[16] K. Ueda and M. Morita. Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, May 1994. In Press.

A Detailed Pseudocode

This appendix gives detailed versions of the pseudocode given in Figures 6, 7, 8, and 9. The question of locking is addressed, and the details of message creation and storage are given. It is hoped that this sketch is in sufficient detail to be implementable in a fairly straightforward fashion.

To switch threads:

- Lock `ready_queue.lock`.
- Dequeue a message `m` from `ready_queue`.
- Unlock `ready_queue.lock`.
- Place `m.mr` in `mr`.
- Place `m.fp` in `fp`.
- Jump to `m.ip`.

To **read** a variable `V` passed in `mr` in a procedure `p`:

- Lock `mr.lock`.
- If `mr.state` is `written`:
  - Unlock `mr.lock`.
  - Switch threads.
- If `mr.state` is `read`:
  - Unlock `mr.lock`.
  - Find the binding index `bi` denoted by `mr`.
  - Place `bi` in `mr`.
  - Jump to `p.bind`.

To **bind** an output argument of a procedure `p`:

- If `mr` refers to a tell argument:
  - Switch threads.
- If `mr` refers to a call argument:
  - Find the procedure `q` and binding index `bi` denoted by `mr`.
  - Find the call index `ci` of `q`.
  - Place `bi` in `mr`.
  - Lock $\texttt{fp.call}_{ci}\texttt{.lock}$.
  - If $\texttt{fp.call}_{ci}\texttt{.callfp}$ is non-nil:
    - Place $\texttt{fp.call}_{ci}\texttt{.callfp}$ in `fp`.
    - Jump to `q.bind`.
  - If $\texttt{fp.call}_{ci}\texttt{.callfp}$ is nil:
    - Load $\texttt{ap}_0..\texttt{ap}_n$ with `q`'s arguments.
    - Allocate a new frame `f`.
    - Lock `f.lock`.
    - Place `f` in `fp`.
    - Place $\texttt{call}_{ci}$ in `cr`.
    - Jump to `q.create`.

To **create** a new frame (in `fp`) and bindings in a procedure `p`:

- Initialize `fp.commit_index` to `uncommitted`.
- Initialize `fp.tobind` to a list containing `mr`.
- For $i = 1..n$ where n is the number of variable table entries:
    - Initialize $\texttt{fp.var}_i$ to <`unlocked`, `unread`, <`fp`, `p.read`>>.
- For $i = 1..n$ where n is the number of body calls:
    - Initialize $\texttt{fp.call}_i$ to <`unlocked`, `nil`>.
- Place `fp` in `cr.fp`, marking the frame as created.
- Unlock `fp.lock`.
- Unlock `cr.lock`.
- Jump to the decision graph for `p`.

To **suspend** decision graph evaluation in a frame `fp`, resuming at address `ra`:

- Set `fp.suspcount` to the number of variables being suspended on.
- For each variable `V` whose binding is needed:
    - Lock `V.lock`.
    - If `V.state` is `unread`:
        - Save the `V.binding.fp` in some register `vfp`.
        - Save the `V.binding.ip` in some register `vip`.
        - Set `V.binding` to a list containing <`fp`, `ra`>.
        - Set `V.state` to `read`.
        - Create a new message `m`.
        - Set `m.fp` to `vfp`.
        - Set `m.ip` to `vip`.
        - Set `m.mr` to `V`.
        - Add `m` to the `ready_queue`.
    - If `V.state` is `read`:
        - Add <`fp`, `ra`> to `V.binding`.
    - Unlock `V.lock`.
- Switch threads.

To **resume** a suspended decision graph:

- Decrement `fp.suspcount`.
- If `fp.suspcount` is non-zero, switch threads.
- If `fp.suspcount` is zero, resume decision graph execution.

To **commit** to a particular clause in a frame `fp`:

- Lock `fp.lock`.
- Set `fp.commit_index` to the desired clause.
- For each tell output argument `V`:
    - Lock `V.lock`.
    - If `V.state` is `read`:
        - Save the binding of `V` in some temporary register `r`.
        - Set `V.state` to `written`.
        - Set `V.binding` to its new value.
        - Unlock `V.lock`.
        - For each continuation `c` in `r`:
            - Add <`c.fp`, `c.ip`, `nil`> to the `ready_queue`.

        - If `V.state` is `unread`:
            - Set `V.state` to `written`.
            - Set `V.binding` to its new value.
            - Unlock `V.lock`.
    - For each call output argument `V`:
        - Lock `V.lock`.
        - If `V.state` is `unread`:
            - Set `V.binding` to `p.read`.
        - Unlock `V.lock`.
- For each non-tell binding index `bi` in `fp.tobind`:
    - Add a message <`fp`, `p.bind`, `bi`> to the `ready_queue`.
- Switch threads.