The Diadora Principle: Efficient Execution of Concurrent Logic and Related Languages

B.C. Massey and E. Tick

CIS-TR–93–15
June 1993

Abstract

Compile-time partitioning of fine-grained concurrent languages is difficult, because it must be both powerful enough to greatly increase the grain size and safe from introducing errors, particularly cycles (data dependencies that circularly link tasks). Incorrect assignment of a cycle into a single thread can result in deadlock, but perfectly safe data dependency analysis is especially difficult for concurrent logic programming languages (CLPs) because of the prevalence of logic variables, which can cause hidden cycles through aliasing, and because efficiency requires that many tasks be placed in the same thread.

This paper describes the Diadora computation model, developed to partition CLPs by instituting *deadlock breaking* if an inadvertent cycle has been sequentialized, and *task stealing* for runtime load balancing, thus exploiting the synergy between simple static analysis and a simple runtime system. The Diadora model is a form of lazy task creation *a la* Mohr, Kranz and Halstead. However, it is customized for CLPs and the somewhat different problems and features of these languages.

This paper was presented at*Hawaiian Internation Conference on Systems Sciences,* Wailea Maui, January 1994.

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

Contents

1 Introduction

A primary difficulty in achieving high-performance implementations of concurrent languages is creating optimal task granularities. If the language is inherently fine-grained, as is for instance Id or Flat Concurrent Prolog [16], spawning parallel tasks, or even concurrent tasks residing on the same processor, becomes expensive because of both communication and task management overheads. These issues are outlined in numerous papers for both functional and logic programming languages, and the technique of *partitioning* has been shown to assist in "granularity collection" at compile time [8, 18].

The difficulties in partitioning a program are numerous. Primarily, data dependencies can cause "cycles" among tasks, which preclude sequentializing or threading the tasks. To avoid deadlock, a safe data dependency analysis must be conducted at compile time. This implies global analysis (by this we mean across procedure calls) in concurrent logic languages because of the flexibility of the logical variable: aliasing can occur anywhere in the program. Functional languages have analogous problems with non-strictness, wherein redexes must be evaluated concurrently. Aliasing is in general less prevalent in functional languages, although of course array references are notorious trouble spots.

Most partitioning algorithms for functional languages do local analysis within a single procedure. This may be sufficient for dataflow languages where procedure bodies have iterative loops within them, although even here it is not be enough to achieve very high performance, as evidenced by Traub and Culler's [18] seminal work on global analysis. In concurrent logic languages, such local analysis is surely not sufficient because clauses consist entirely of procedure invocations: there is no iterative construct.

Our goal was therefore to develop both an inexpensive (at compile time) and effective method of partitioning such fine-grain languages. Others working in this area include Mohr, Kranz and Halstead [13], who invented *lazy task creation* in the context of concurrent Scheme, and King and Soper [8], who proposed a global abstract interpretation algorithm for deriving safe data dependency information about concurrent logic languages. Lazy task creation as implemented in [13] must handle more difficult cases in order to work with a more complicated language implementation than is typical of CLPs. Thus, either stack copying or complicated stack structures are required in order to "steal" tasks: a complexity we can avoid by the nature of our language implementation. King and Soper's partitions are always correct in the sense that cycles are conservatively kept out of threads. We feel that there is little point in paying a steep compile-time cost to derive safe information if at runtime everything is naturally safe. Furthermore, in an effort to derive only safe information, conservative abstractions lose information, potentially resulting in less granularity than can theoretically be exploited.

In this paper we describe a simpler mechanism than both, called the Diadora model, that builds good partitions using not necessarily correct information. The point is that in reality, few programs have complex "hidden" cycles. We further describe what we mean by hidden cycles in Section 3.1, but intuitively they are cycles that are not syntactically apparent, e.g., within a single clause.

The paper is organized as follows. Section 2 describes the Diadora principle. Section 3 proposes an implementation of the model. Section 4 describes our progress in implementing the proposed model. Section 5 summarizes the literature in this area. Conclusions are given in Section 6.

2 The Diadora principle

In this section we first review the informal semantics of concurrent logic programming languages [16] and then explain the Diadora principle in this context. The model is general enough to be applied to non-strict functional languages also. A detailed discussion of implementation issues is given in the next section.

A committed-choice logic program is a set of guarded Horn clauses of the form:

$$H \text{ :- } G_1, \ldots, G_m \mid B_1, \ldots, B_n$$

where $m, n \geq 0$. H is the clause head, G_i is a guard goal, and B_j is a body goal. The commit operator '$|$' divides the clause into a passive part (the *guard*) and active part (the *body*). "Flat" languages have a further restriction that guard goals are simple builtin functions, e.g., $\leq$. A goal g commits to a clause p, if g successfully *matches* with the head of clause p (i.e., without causing any bindings to the variables of the goal) and the guards of clause p succeed without binding any goal variable. When a goal can commit to more than one clause in a procedure, it commits to one of them nondeterministically (the other candidates are thrown away).

Structures appearing in the head and guard of a clause cause *suspension* of execution if the corresponding argument of the goal is not sufficiently instantiated. A suspended invocation may be *resumed* later when the variable associated with the suspended invocation becomes sufficiently instantiated. A program successfully terminates when, starting from an initial user *query* (a conjunct of atoms), after some number of reduction steps, no goals remain to be executed, nor are suspended.

For example, consider the program of Figure 1. When invoked via the query $p(C, D)$ it returns the bindings $C = 1$, $D = 1$.

$$\begin{array}{l} p(T,V) \text{ :- } true \mid \\ \quad q(1,S,S,T), \\ \quad q(U,V,1,U). \\[1ex] q(W,X,Y,Z) \text{ :- } true \mid \\ \quad r(W,X), \\ \quad r(Y,Z). \\[1ex] r(A,B) \text{ :- } int(A) \mid \\ \quad B = A. \end{array}$$

Figure 1: An FGHC program

A meta-level sequentialization operator, *'&'/2*, as defined in Parlog [2], specifies that its two operands are to be executed in sequential order. One can "sequentialize" an entire program by replacing all parallel conjunctions *','/2* with sequential ones (although this will likely slow down the program in current Parlog implementations because checking for termination of a sequential goal has overheads). For simplicity, we can imagine a normal clause form wherein body goal conjunctions are all parallel or are all sequential. Rewriting to normal form is trivial: new procedure invocations are introduced. Furthermore, we place an artificial restriction on the program that no sequential goal can be defined with parallel goals. In other words, if we produce a DAG by collapsing the strongly connected components of the call graph, then only full subtrees in this DAG can be sequential (a full subtree contains all children deriving from a single root). This restriction is not necessary, but it simplifies the explanation of the model, and is practically useful because it engenders efficient execution of sequentialized goals.

The program of Figure 1 (an example due to King and Soper [8]) is particularly interesting in this regard. If one sequentializes $q/4$ by simply replacing the *','/2* with *'&'/2*, the program will deadlock during the second call to *q/4*, since A will never be bound in the first call to *r/2*. One might try reversing the order of the calls to *r/2* in the sequentialized program, which will indeed keep the second call to *q/4* from deadlocking. However, the *first* call to *q/4* will now deadlock. In short, no static sequentialization of *q/4* is safe, and yet there is nothing about *q/4* itself which suggests that this is the case—the unsequentializability of *q/4* is a *global* property of the program. If one constructs a DAG representing the dependencies between the goals in Figure 1, a *cycle* will appear. It is the incorrect sequentialization of these cycles which is the fundamental error which must be avoided in static analysis.

We assume that static analysis produces a data dependency graph that indicates the dependencies between goals in the program (this is discussed in Section 3.1). This analysis will indicate which parallel conjunctions are necessary, and which sequential conjunctions are desirable (considering granularity tradeoffs, e.g., [8, 15]). Whatever the information, a normal form program can be created and code generation can proceed to compile the threads, i.e., subtrees. A given thread produces standard stack-based procedural invocation code sequences. A goal is invoked by creating a frame of its arguments on the top of its associated stack. It will invoke its body goals in sequential order by creating frames for each one in sequence. Any of them can utilize the stack and return. We will not go into details here, but standard optimizations, such as exploiting registers, can be used. For simplicity we view all relevant, dynamic data structures as potentially resident on a global heap.[1] Thus a procedure returns values via the heap.

Clauses consisting of parallel conjunctive goals are compiled in a manner similar to most CLP language implementations (e.g., [4, 7, 17]). This means that each body goal is wrapped up in a goal record and placed in a ready set for work distribution. Usually one of the goals is chosen for immediate execution (a weak form of tail-call optimization).

A task in the Diadora model consists of a stack of frames rather than a goal record. In the degenerate case of a parallel body goal defined by clauses of further parallel body goals, the stack is a single frame. A unique element of the frame is the break continuation: a code label to be executed to "break" the frame's invocation into several parallel subtasks. The idea is similar to Mohr, Kranz and Halstead's lazy task creation [13]. The break

[1]This has implications with respect to the lazy task creation model [13] — see Section 5.

continuation for a degenerate stack (essentially a goal record) is empty.

The Diadora model proceeds by reducing goals sequentially, assuming some type of scheduling that distributes the execution among the processors. Synchronization occurs when a goal invocation requires a value for an input variable that has not yet been bound. In this case, the goal is suspended. This might be implemented by *hooking* the required variable to the goal. Note that even on a uniprocessor, suspension must still be facilitated in some manner. For example, compilation of CLPs into Prolog can be easily done if the Prolog offers a *freeze* predicate, effectively doing the variable hooking and resumption. A key point is that subsequent binding of that variable will eventually resume the hooked goal, i.e., add it to the ready set.

We now give a brief overview of the execution of a sequential thread and how the continuation is managed. This sketch is detailed in the next section. Consider a goal: "a :- b & c & d & e." Assume that a is itself in a parallel conjunction, so that a new stack is created for it, with the bottom frame holding its arguments. The initial break continuation points to the code segment: "c, d, e." representing the concurrent invocation of these body goals. If b returns successfully to a (without suspending), then a modifies its break continuation to point to the code segment: "d, e." and so on. Notice that these code segments overlap so that we need only generate two copies of the clause: a sequential and a parallel version. The continuations then become labels between the parallel goals. (However, in the presence of traditional compiler optimizations of the sequential version, some "glue" code may be required for each continuation.)

If the ready set is empty and no goals are suspended then the program terminates. If the ready set is empty but one or more suspended goals remain, then we attempt to break the deadlock, similar to Andorra and Pandora [3, 1]. Deadlock is broken by selecting some goal, near the top of a stack, that has a non-empty break continuation. Picking a top goal is a heuristic aimed at quickly breaking deadlocks due to cycles incurred near the present invocation.

Consider that if b suspends, and this stack is eventually chosen to break deadlock, the break continuation to "c, d, e." will be executed. Furthermore, the return continuation in the frame of b must be modified to return to the end of a, not to the call of c (since c, d and e have already been spawned).[2] After injecting these three additional goals into the ready set (each which has an empty continuation), deadlock might or might not remain. In the steady state, the model coroutines between *running* and *breaking*, i.e., between a state of normal execution and a state of deadlock breaking.

In summary, several points should be emphasized about the model. First, no backtracking is needed to break deadlock because these are committed-choice languages, i.e., all bindings produced are valid in any possible execution of the given program. Second, the overhead of breaking deadlock is relatively small: we jump to a continuation which performs roughly the same code that would have been executed in a fully concurrent model. Little extra recomputation is performed. Third, and critically, deadlock is still costly because it causes all processors to idle. Even detecting this situation can be costly. Note that in a standard model, deadlock detection is not critical because it is true deadlock, i.e., there is

[2]Variations of this scheme are of course possible. For example, we could spawn b, c, d, and e and rewrite a's stack appropriately.

no way to break it and thus it is a programmer error.

We believe, however, that cycles causing deadlock are both rare and local. By the latter, we mean that breaking the deadlock from the top-of-stack down will be an efficient strategy. If both of these hypotheses are correct, the impact of deadlock on performance should be fairly low.

3 Implementation issues

In this section we describe the static analysis, code generation and abstract machine design, and runtime system support needed to successfully implement the Diadora model.

3.1 Static analysis

As noted earlier, in order to avoid frequent deadlock, we need precise static information about dependencies between procedures. Fortunately, there are several standard analyses which provide such information at reasonable cost. A basic strategy [12] is to first identify which variables in a clause are bound by the caller, and which by the callee. Once these "modes" have been determined, the induced graph of dependencies between body goals gives the information necessary to sequentialize the program. Finally, threaded code is generated based on the global information.

We benefit from restricting ourselves to subset of the CLP family that allows full mode information to be derived. Our choice is Fully-Moded Flat Guarded Horn Clauses (FGHC) defined by Ueda and Morita [19], representing a class of languages such as Strand [5] and Janus [14]. We call this language FM for short. The essential idea of FM is that the input/output structure of corresponding clause arguments of a procedure is required to be identical. Thus, for example, the procedure

$$p(a(X), b(Y)) \ :\!- \ X := Y + 1.$$

$$p(a(X), b(Y)) \ :\!- \ Y := X + 1.$$

is not moded, since the first subterm of the first argument of the first clause of p is *output* (i.e., is bound by the clause), whereas the first subterm of the first argument of the second clause of p is *input* (i.e., may be bound by the caller). Note that in a language such as Parlog, all arguments of all clauses of p would be *input*. In other words, Ueda and Morita's mode analysis details the entire input/output structure, not just the top level. The phrase "the first subterm of the first argument of the first clause" is an example of a *path*, noted as $\{< p, 1 >, < a, 1 >\}$. Each path has a mode of *output* if any variable occurring at that path position is bound in the procedure body, or of mode *input* otherwise. Because of recursion, there is potentially an infinite set of paths for each procedure, all of which need to be moded for the procedure itself to be fully moded.

Once we have moded all possible paths of all procedures in a program, we can immediately and precisely derive all local dependencies between the invocations of procedures in a clause body. Ueda and Morita outline a scheme whereby all mode information can be derived (and thus the fully-modedness of a program checked) by unification of rational trees. Essentially, some paths in a clause are found to be input by identifying that the argument must be bound externally for the clause to be executable, and some paths are found to be output by identifying direct bindings of them in the body of the clause. Then, since all clauses of a given procedure must have the same mode for each path starting in that procedure, unification may be used to combine the mode information from the various clauses, and from the body goals of those clauses.

We have implemented this algorithm and an alternative mode analysis algorithm based on a finite set of paths [11]. The former analysis provides precise information whereas the latter analysis cannot detect the non-modedness of some FGHC programs. However, the latter algorithm has the comparative advantage that it is easy to implement and runs quickly. Because the finite-path algorithm is not complete, it might produce inaccurate mode information which could lead to deadlock through incorrect sequentialization of an FGHC program.

Though we find that in practice most of the modes of real-world FM programs are derived by the algorithm, and that most of the modes which are derived for real-world FGHC programs are correct, the possibility that a compiler dependent on this mode analysis algorithm would be unable to generate code or would generate incorrect code in some cases inhibits its apparent usefulness. Fortunately, incorrect sequentialization of a program under the Diadora model never affects the correctness of the program, merely its efficiency. Thus, our finite-path mode analysis algorithm may be safely used to sequentialize FGHC programs for use under Diadora.

Still another approach is to apply abstract interpretation, either locally or globally. Perhaps typical of such schemes is King and Soper's "producer-consumer analysis" [8, 9]. The main advantage of such a method is that it produces safe results for non-fully-moded programs. The main disadvantage is that it appears to be complex to implement and expensive to execute. We are currently unaware of any published empirical studies of *any* of these static analyses for comparison.

Once we have accurate information about the producers and consumers of all variables in the program, we topologically sort the induced dependency graph in order to obtain a threading of the program. The catch is that the dependency graph may contain cycles! These cycles represent *feedback*, i.e., the use of the output of a subprogram invocation as an input to that invocation. Note that this differs from recursion, in which the output of one subprogram invocation is used as the input to another invocation of the same subprogram.

In the absence of complete dependency information, it may be impossible to decide whether feedback is present in a procedure or not. It is always possible to take the safe approach of breaking all such potential cycles across multiple threads. This is the approach taken by Traub and Culler [18] for functional languages, and by King and Soper [9] for CLP languages.

Even in the Diadora model, it is beneficial to break known cycles at compile time. This avoids the runtime overhead of deadlock and breaking such cycles. It is debatable whether to break potential cycles at compile time, however, since it depends on the tradeoff between the benefits of runtime sequentiality and the costs of dynamically breaking deadlock, weighed by the estimated probability that the deadlock will occur.

3.2 Compilation

In this section we explain the abstract machine and corresponding code generation we envision for the Diadora model. Once we have a sequentialization for the body goals of a program, we may generate machine code for its execution. For each procedure, the compiler will generate code to handle the head tests, and select an appropriate clause body, code to sequentially execute the body goals and primitives, and alternative code to spawn the body goals and primitives concurrently (if necessary). Lastly the invocation returns.

The abstract machine model consists of multiple workers, each of which executes tasks and shares a global address space called the heap. A task contains a program counter, a lock (for synchronization during deadlock breaking), a stack and pointers to its base and top. The stack contains frames which are abstractly tuples of the form <*return*, *break*, *contextp*>. *Contextp* points to an n-tuple of invocation context on the heap. Consider the example shown in Figure 2, annotated by the compiler with proposed sequentialization operators &. The compiler might produce pseudo-code for b as shown in Figure 3.

Although this is unoptimized code for a simple example, it should be sufficient for understanding the basic concepts. The pseudo-code consists of six basic blocks. The first block corresponds to the head of b and the call to the first body goal c. The second block handles the return from c, the updating of the break address of b, and the call to j_1, all on a locked stack to avoid races with potential task stealers. The third block is analogous to the second, except that it calls j_2. The fourth block merely returns from b. The fifth block is the break continuation which spawns the second body goal j_1 of b as a separate task during deadlock breaking or task stealing. It falls through into the sixth block, which spawns j_2. Thus, the code is about twice as large as its traditional counterpart would be, since for each basic block in the sequential version, there is a corresponding parallel block.

Note that popping the topmost stack frame on return from a procedure call is an action of the caller rather than the callee (lazy task creation does likewise [13]). This is because of the invariant that the stack must be locked in order to pop a stack frame, which prevents a race between the current worker and another worker attempting to steal a task from it.

3.3 Runtime support

One of the benefits of the Diadora model is that it doesn't require a great deal of runtime support. Conceptually, the runtime system for a Diadora implementation consists of an active set, a ready set, and a suspended set. In each set is zero or more tasks as described previously. A global heap is necessary for resizable, persistent, and shared data objects, although all data objects may be stored there in simple implementations.

In particular, there is no requirement for an explicit task manager in Diadora. Instead, the compiler may generate code and runtime system calls such that each task does its own manipulations of the runtime data structures. Thus, the code in the continuations creates and activates new tasks, and each task contains code to perform its own garbage collections, and to delete itself. This allows great efficiency for the Diadora "runtime system," since each task typically can manipulate its own state cheaply and without the possibility of race conditions.

$$
\begin{aligned}
a(X_1, X_8) &\text{ :- } b(X_1, X_8). \\
b(X_1, X_2) &\text{ :- } c(X_1, T_1) \;\&\; j_1(T_1, T_2) \;\&\; j_2(T_2, X_2). \\
c(X_1, X_3) &\text{ :- } d(X_1, X_3). \\
d(X_1, X_4) &\text{ :- } e(X_1, T_3) \;\&\; k_1(T_3, T_4) \;\&\; k_2(T_4, X_4). \\
e(X_1, X_5) &\text{ :- } f(X_1, X_5). \\
f(X_1, X_6) &\text{ :- } g(X_1, T_5) \;\&\; m_1(T_5, T_6) \;\&\; m_2(T_6, X_6). \\
g(X_1, X_7) &\text{ :- } h(X_1, X_7).
\end{aligned}
$$

Figure 2: Sequential FGHC program

b: set this frame's break address to b2p
    push a new frame <b2, empty, $< X_1, T_1 >$>
    jump to procedure c
b2: lock this task's stack
    set this frame's break address to b3p
    pop old frame
    push a new frame <b3, empty, $< T_1, T_2 >$>
    unlock this task's stack
    jump to procedure j_1
b3: lock this task's stack
    set this frame's break address to empty
    pop old frame
    push a new frame <b4, empty, $< T_2, X_2 >$>
    unlock this task's stack
    jump to procedure j_2
b4: jump to this frame's return address
b2p: create a new task
    give new task frame <exit, empty, $< T_1, T_2 >$>
    set the new task's program counter to j_1
    mark the new task active
b3p: create a new task
    give new task frame <exit, empty, $< T_2, X_2 >$>
    set the new task's program counter to j_2
    mark the new task active
    set this frame's break address to empty
    jump to this task's saved program counter

Figure 3: Pseudo-code for sequential program

g	f2	empty	$< X_1, X_7 >$
f	e2	f2p	$< X_1, X_6, T_5, T_6 >$
e	d2	empty	$< X_1, X_5 >$
d	c2	d2p	$< X_1, X_4, T_3, T_4 >$
c	b2	empty	$< X_1, X_3 >$
b	a2	b2p	$< X_1, X_2, T_1, T_2 >$
a	exit	empty	$< X_1, X_8 >$

(a) Before deadlock breaking

g	**f3**	empty	$< X_1, X_7 >$
f	e2	**empty**	$< X_1, X_6, T_5, T_6 >$
e	d2	empty	$< X_1, X_5 >$
d	c2	d2p	$< X_1, X_4, T_3, T_4 >$
c	b2	empty	$< X_1, X_3 >$
b	a2	b2p	$< X_1, X_2, T_1, T_2 >$
a	exit	empty	$< X_1, X_8 >$

(b) After deadlock breaking

Figure 4: Stack snapshot during program execution

Returning to our example, some of the runtime system calls in the pseudo-code of Figure 3 need additional explanation. The "exit" continuation inserted into the stack frame by the generated code is a special continuation which will deallocate a task (probably implicitly, via the garbage collector), and then pick up another task. The action of "picking up another task," used by the exit continuation as well as other runtime routines such as suspension, is a bit complicated. To pick up another task, one chooses the first action which applies, from the following list:

1. Check the ready set for tasks and jump into one if found.
2. Scan the suspended set, looking for a task whose hooked variable is now bound. Jump into one if found.
3. Scan the suspended set, looking for cycles of tasks suspended on each others' variables. If such a deadlock is found, break it as described below.
4. Pick a task from the active set and steal new tasks from it as described below.
5. No tasks are active, ready, or suspended, then the program is finished. Halt.

The "deadlock breaking" method is simple. Note that if no task in the cycle has a non-empty break continuation, anywhere on its stack, then the deadlock is real, and a program error is signaled. Otherwise, one calls the break continuation of some task in the cycle which is nearest to the top of the stack. This creates a new set of active tasks, one of which will hopefully supply the bindings needed to get execution started again. The choice of the topmost break continuation is mainly because of the hypothesis, mentioned previously, that most deadlocks are local. It is important to free deadlocked producers as quickly as possible, so that consumers may be restarted as well.

Consider our sequentialized program of Figure 2 at a point just after the execution of g. At this point, given that the compiler generates labels similar to those of Figure 3 for each procedure of Figure 2, the upward-growing task stack will look like Figure 4a. If we were to deadlock at this point, we would choose to break the deadlock via the break address $f2p$ in the frame f. We replace the return address in g's frame with $f3$, the ending address of f, replace the break address of f with empty, and invoke the old break address of f ($f2p$) to create new subtasks. The original task's stack after breaking is shown in Figure 4b.

The "task stealing" method is a bit trickier. This is because the task from which tasks are being stolen is currently active. If interrupts are cheap, the system may be simplified by forcing a task to suspend before stealing from it, but this is not normally the case. Thus, before stealing from a task, we lock the targeted task's stack, which ensures that the active worker may not return through the frame we are trying to steal from, since the worker locks the stack before each return. This adds overhead to the return relative to a conventional language implementation. However, the overhead of task termination via locked return is still low compared to the overhead of traditional CLP implementations, which must select a new task from the active set at every task termination point.

We conclude this section by listing a few potential improvements to the previously described schemes:

- The compiler may choose to place variables on the stack of their creating task, rather than on the heap, to reduce the burden of the garbage collector and the virtual-memory system. This complicates the implementation greatly, but could be of great benefit for allocator-intensive applications. However, this would require the stack manipulations of the lazy task creation model [13], as described in Section 5.

- Even stack-based variables in CLP implementations are logic variables, which means that the overhead of synchronization (usually by testing for boundness) is incurred on every read operation, and potentially on every write operation. Although somewhat orthogonal to Diadora, it would be nice to replace some of the logic variable implicit synchronization in the compiled code with explicit synchronization and "ordinary" variables, in order to improve code efficiency. Unfortunately, this requires a compiler to perform complex static analysis of a sort that is not currently well-understood, and thus needs further research to be practical.

- Various standard intraprocedural optimizations may be performed on the sequential version of the code, and indeed some interprocedural ones. The only constraint is that no optimization should be performed which irretrievably loses runtime data needed by the parallel code, e.g., by destroying vital context.

- The order in which the suspended, active, and ready sets are scanned by the above algorithms is very important. It is likely to be possible to prioritize these sets so that the scan time is near minimal.

- The sample pseudocode suggests a policy in which breaking off concurrent goals is done *eagerly*; once the break continuation is called, all remaining goals are spawned. It is possible, and may be desirable, to implement other policies. For example, if the following lines are added to the end of block $b2p$ in Figure 3

  ```
  set this frame's break address to b3p
  jump to this task's saved program counter
  ```

  then a call to $b2p$ will only break off the first concurrent goal j_1, and a second call to $b3p$ will be required to spawn j_2. This may be preferable in order to minimize the number of extra goals spawned in a break operation, although it inhibits amortizing the cost of the break operation across multiple new goals.

- A good choice of which task to parallelize when deadlock is detected may be crucial to the efficiency of the Diadora method, although it is not necessary for its correctness. The important point is that if one randomly chooses goals to parallelize at deadlock, the performance should be similar to that of a simple fully-concurrent model.

4 Morocco

Some progress has already been made in implementing the system described above. We expect very shortly to complete Morocco, a system consisting of a translator which will convert FM programs into C language code, and a collection of C language runtime routines which will be called by the emitted code.

This work reuses much of the implementation of Monaco, a high-performance CLP language system for shared-memory multiprocessors [17]; in particular, the Monaco compiler front-end is reused relatively intact. This allows us to leverage the collection of clauses into procedures and the complex implementations of incremental passive unification and decision-graph generation [10] from the Monaco front-end.

Fundamental to the implementation of Diadora is the ability to access and modify stack values, including parameters, temporaries, the return address, and the break address. This would seem to rule out a pure C language implementation, as the C stack is essentially inaccessible from C code.

Fortunately, it is possible to work around this problem by using the `_setjmp` and `_longjmp` ANSI C library routines, which allow the user to save into a `jmp_buf` something which is effectively a continuation. Thus, the operation in the pseudocode of saving an address corresponds to calling `_setjmp`, and jumping to an address is done by `_longjmp`. Each FM procedure can thus be translated into a corresponding C procedure.

Each C procedure so produced has an extra "hidden" argument, which is used to pass in extra information from the caller. This information is linked together in a linked list which acts much like the C stack, allowing stack traversal during breaking and stealing.

We expect that Morocco will exhibit performance competitive with existing implementations of fine-grained concurrent languages. It will certainly serve as a useful model system, which will help us evaluate some of the potential improvements to Diadora outlined above before attempting a Diadora-based native-code FM compiler.

5 Literature review

In this section we primarily review the work of Mohr, Kranz, and Halstead [13] on *lazy task creation*, which is closest to the Diadora concept. Their work is based on a mostly-functional concurrent dialect of Scheme and shares the motivations of our research: increasing the granularity of too-fine programs. Their model is more or less a superset of Diadora because it can also handle implicit eager procedure calls and full functional languages. The terminology of task stealing and deadlock breaking is originally theirs, as is the idea of locking the stack to guard against races during stealing.

In comparison with Diadora, the lazy task creation implementation is complicated by the problem of downward stack references: any stack frame may contain references to objects which are in frames nearer the bottom of the stack. When creating a new task with a given stack frame as context, care must be taken that downward references contained in that frame remain valid. Mohr, Kranz, and Halstead thus present two implementations, one which copies stack frames, and another which uses a linked list of heap-allocated frames rather than a traditional stack. Since CLP language implementations typically allocate all variables on the heap, these potentially expensive complications are not necessary for Diadora, which can merely modify the existing stack and create new stacks from scratch. However, if we plan to keep logic variables directly in stack frames, then we need to move to a model like that of [13].

Their implementation is also complicated by the fact that only explicit concurrency via *futures* is permitted in their target language, so a special *lazy task queue* must be maintained to record the occurrence of deactivated futures. A future can create only a single concurrent task, so the more general concept of breaking a frame into multiple tasks is unnecessary. This implies that the runtime system, rather than the compiler, contains the code which does task creation.

Actual timings for two implementations of lazy task creation show that the performance is quite good for the target language [13]. We fully expect these performance improvements to carry over to CLP languages, especially given the extremely fine-grained parallelism of traditional implementations.

As we have reviewed throughout the paper, work related to the process of compiling for Diadora includes the work of King and Soper [8] on producer-consumer analysis, Ueda and Morita's [19] work on mode analysis, and the work of Traub and Culler [18] on forming threads. All of these are, however, conservative, in the interests of safety for traditional implementations. Thus they form smaller threads than might be produced given the deadlock-breaking ability of Diadora/lazy task creation. Our model is close in spirit to the Andorra model [3], with deadlock occurring in Andorra due to favoring determinate procedure invocations, whereas deadlock occurs in Diadora due to favoring sequential threads. An open research issue is how the models compare in terms of compilation effort and static code size, and dynamic performance.

6 Conclusions

The paper introduced a new parallel execution model for concurrent logic programs (CLPs) based on dynamic deadlock breaking and lazy task stealing of threaded code. In the logic programming community, these concepts are novel, although they were originally introduced in another form for concurrent functional languages by Mohr, Kranz and Halstead [13]. The Diadora model differs from the previous work in that 1) CLPs are implicitly concurrent, increasing the threat of deadlock, and therefore our model emphasizes the synergy between good static analysis and dynamic deadlock breaking; 2) Heap-based CLP implementations allow simpler stack management; and 3) the interaction between deadlock breaking and task stealing in the extremely fine-grained (far more than futures-based languages) CLP environment requires innovative solutions to various implementation problems.

In summary, the advantages of the model include:

- Diadora acts as a "morning-after pill" for CLPs, allowing for unsafe information to be utilized efficiently by supporting a trapdoor in the rare case that cycles deadlock the computation. Safe analysis for sequentialization is very difficult to do quickly and accurately. Fundamentally there is a tradeoff during static analysis between speed and information. In general, global static analyses for CLP languages are slow. Furthermore, to ensure that they are safe, conservative estimations usually degrade the quality and utility of the information that they produce. The application of data dependencies is a particularly degenerate situation because it is almost never the case that critical cycles exist in a computation. Noncritical cycles can appear locally, e.g., when using difference lists in a divide-and-conquer all-solutions search, but they do not cause deadlock.[3]

- Diadora exploits synergies between static and dynamic approaches. Fine-grain languages at this level require partitioning across procedure calls to enable construction of large-enough tasks to overcome communication-to-computation ratios. This opens the door to aliasing problems and the myriad of potential, but improbable, cycles they bring. After much thought, it became clear to us that no simple static or dynamic approach could guarantee both safety and efficiency.

- Diadora does not support backtracking, and as a result retains a lean abstract machine, even though it incorporates running/breaking coroutining. Considering the Andorra model, one representative implementation, Pandora [1], is quite complex compared to its JAM [4] origin. This is because of the variable extensions needed in the storage model to support backtracking.

- Diadora does not produce unduly large or complex programs. Typically, a Diadora-compiled program will be about twice the code size of its traditional counterpart, and the excess will consist largely of simple straight-line code.

Potential disadvantages include too-frequent deadlocks and high runtime overheads. Empirical studies are necessary to evaluate these threats. Thus our future work includes implementing the model. The first implementation will be as described above in Section 4. The first empirical studies will be to compare the performance results with those reported for Mul-T [13] and Andorra [3].

Acknowledgements

This research was supported by an NSF Presidential Young Investigator award, with matching funds provided by Sequent Computer Systems Inc., and by a research grant from ICOT. Thanks to Saumya Debray and others for excellent discussions about the concept and implementation.

[3] These dependences are sometimes called non-strictly independent [6].

References

[1] R. Bahgat and S. Gregory. Pandora: Non-deterministic Parallel Logic Programming. In *International Conference on Logic Programming*, pages 471–486. Lisbon, MIT Press, June 1989.

[2] K. L. Clark and S. Gregory. PARLOG: Parallel Programming in Logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, January 1986.

[3] V. S. Costa, D. H. D. Warren, and R. Yang. The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model. In *International Conference on Logic Programming*, pages 825–839. Paris, MIT Press, June 1991.

[4] J. A. Crammond. The Abstract Machine and Implementation of Parallel Parlog. *New Generation Computing*, 10(4):385–422, August 1992.

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

[6] M. V. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *International Conference on Logic Programming*, pages 237–252. Jerusalem, MIT Press, June 1990.

[7] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and its Instruction Set. In *International Symposium on Logic Programming*, pages 468–477. San Francisco, IEEE Computer Society Press, August 1987.

[8] A. King and P. Soper. Heuristics, Thresholding and a New Technique for Controlling the Granularity of Concurrent Logic Programs. Technical Report CSTR 92–08, Department of Electronics and Computer Science, University of Southampton, 1992.

[9] A. King and P. Soper. Schedule Analysis of Concurrent Logic Programs. In *Joint International Conference and Symposium on Logic Programming*, pages 478–492. Washington D.C., MIT Press, November 1992.

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

[11] B. C. Massey. Sequentialization of Parallel Logic Programs with Mode Analysis. Master's thesis, University of Oregon, September 1992. Also available as Technical report CIS-TR-92-18.

[12] 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.

[13] E. Mohr, D. A Kranz, and R. H Halstead Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Computing*, 2(3):264–280, July 1991.

[14] 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.

[15] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* MIT Press, Cambridge MA., 1989.

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

[17] E. Tick and C. Banerjee. Performance Evaluation of Monaco Compiler and Runtime Kernel. In *International Conference on Logic Programming*, pages 757–773. Budapest, MIT Press, June 1993.

[18] K. R. Traub, D. E. Culler, and K. E. Schauser. Global Analysis for Partitioning Non-Strict Programs into Sequential Threads. In *Conference on Lisp and Functional Programming*, pages 324–334. San Francisco, ACM Press, 1992.

[19] K. Ueda and M. Morita. A New Implementation Technique for Flat GHC. In *International Conference on Logic Programming*, pages 3–17. Jerusalem, MIT Press, June 1990.